Methods for Stratego AI

Iain Found
School of Computer Science
Carleton University
Ottawa, ON, K1S 5B6
iainfound@cmail.carleton.ca

Henry Zhangxiao
School of Computer Science
Carleton University
Ottawa, ON, K1S 5B6
henryzhangxiao@cmail.carleton.ca

Abstract

Stratego is an information incomplete board game with a possible 10^{66} starting arrangements for any new Stratego game and a game tree complexity of 10^{535}, far exceeding the complexity of any other board game. What separates novice level naïvely implemented Stratego bots from human players is the evaluation of the need to perform computations on a particular board state, as well as the introduction of playstyle variability and bluffing. In this paper, we will touch upon five algorithms and their implementations which use varying methods in their setups and executions, as well as their limitations.

1 Data-Based Stratego Bots

One realm of Stratego reinforcement learning relies on the data it gathers from the opponent as well as the environment. Due to the incomplete information nature of Stratego, one of the aspirations for a data-based algorithm is to explore the unknown grid-map while constantly assessing the risk. Expolaratory Stratego bots must also use the information it derives from the playstyle of the opponent to gain an edge in strategical planning.

1.1 Probabilistic Multi-ply Multi-Agent

Using probabilities and Bayesian networks, Stankicwicz has focused on opponent modelling [1] to determine the rank of each piece, based on previous games with the same opponent. The simulated behaviour of the opponent was also considered. A probability distribution is attached to each piece to simulate the behaviour. Modelling the opponent led to correct rank prediction 40% of the time, compared to only 18% in case of not using opponent modelling. This higher prediction has led to a 55% win rate against the opponent without the model of the other player.

1.1.1 Probabilistic Agent

The probabilistic agent evaluates the outcome of all next legal moves and picks the best one. The best move is given by an evaluation function. Since the number of each piece type of the enemy is known, the probabilistic agent stores a table with scores with the remaining pieces. The number of pieces of a particular rank will be associated with that rank in the table. The probability of the agent to choose the rank *r* for the attacked piece of colour $c \in$ {blue,red} is defined as

$$p_r^c = \frac{i_r}{\sum_{s \in R^c} i_s}, \forall r \in R^c$$

where i_r is the number of pieces of rank *r*, the denominator is the number of remaining pieces of the enemy, R^c is the set of all ranks of the pieces the enemy has on the board. However, if it is known that

Preprint. Under review.

Table 1: Comparing Agents using Different Parameters with the Same Setup [1]

#E	red(R)	blue(A)	VR%	VRC%	VA%	VAC%	total	$t_e(m)$
1	random	probabilistic, *fm*	1.4%	75.1%	98.5%	0%	10,000	4
2	random	probabilistic, *fb*	10.2%	10.4%	89.7%	34.27%	10,000	4
3	random	2-ply, *fb*	14.5%	16.5%	85.5%	55.7%	1,000	8
4	random	3-ply, *fb*	16%	6.25%	84%	76.1%	100	21
5	probabilistic, *fb*	3-ply, *fb*	41.1%	42.1%	58.8%	87.5%	450	79

the attacked piece of the enemy was moved, then the immobile pieces are not considered anymore:

$$p^c_{r_m} = \frac{i_{r_m}}{\sum_{s \in R^c_m} i_s}, \forall r_m \in R^c_m$$

where R^c_m is the set of the ranks of the movable pieces the enemy has left on the board, which leads to: $R^{player}_m = R^{player} \backslash \{\text{F,B}\}$.

1.1.2 Muti-ply Agent

Similar to the probabilistic agent, for each legal move, the agent chooses a rank for the attacked piece if there is an attacked piece of unknown rank. Different from the probabilistic agent is that instead of evaluating the resulting position immediately for each move, the multi-ply agent simulates the next moves of the opponent. For this, the player generates a possible game position for the enemy, that is consistent with the information currently known about the opponent. Then it uses the information that it knows the opponent certainly knows about it for keeping the consistency in the newly simulated position. Similarly with the probabilistic agent, the multi-ply agent evaluates the resulting position next and chooses the best move. For simulating the opponent's move, the agent considers if the other agent is a probabilistic or a multi-ply one.

1.1.3 Results with Different Configurations

The agents are compared in Table I, where *total* represents the number of played games. Here, *VR* represents red victories as a percentage of the total, *VRC* are red victories with capture, as a percentage of *VR*. Similarly, *VA* are blue victories as a percentage of the total number of games, while *VAC* are blue victories with capture, as a percentage of *VR*. The execution time is denoted with t_e. Two evaluation functions were used: basic evaluation function *fb* and material evaluation function *fm*. The players use the same configuration for the starting position. [3]

1.2 Invincible - A Stratego Bot

"Invincible" uses original approaches for creating setups, making guesses about the ranks of unknown pieces and reasoning about the best move in a given situation. [2] Setups are created by a combination of statistical information of human-created setups and heuristic evaluation of the results to select the best of a number of semi-random setups. The basis of "Invincible" is a collection of plans that give a value to every possible move. The move with the highest total contribution to all the plans is selected. These plans evaluate only a single ply, so no deeper searches are made. Since plans have a well-defined goal, the complexity of finding how well a move contributes to a plan is usually constant, or at most dependent on the (constant) size of the board.

The advantages and disadvantages of using a plan-based algorithm:

- Advantages:
  - "Unlimited" search depth (4.1).
  - Possibilities to bluff (4.2).
  - Possibilities to coordinate attacks with multiple pieces. (4.3).
  - Focus computation on interesting situations. (4.4).
- Disadvantages:
  - "Invincible" knows only what it's told (4.5).

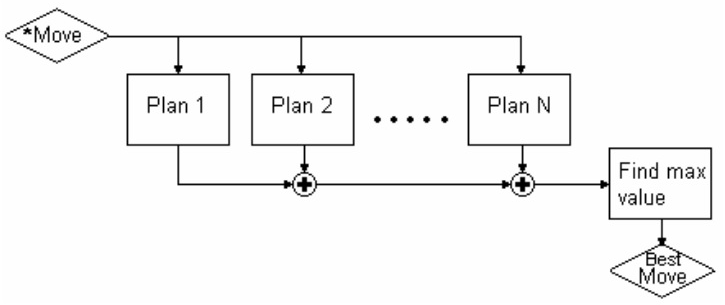

Figure 1: Finding the best move. [2]

- Comparable information is calculated multiple times (4.6).
- Game specific knowledge needed (4.7).

1.2.1 Plans

In particular, plans can see: The current board with the position of all the pieces, the graveyard containing the captured pieces with their rank and color, the dynamically determined values of the pieces on the board, and the history of all moves previously executed. They do not know whether other plans are active in this situation, or even how many other plans there are. Due to this factor, it is important that the values given by plans are realistic in relation to all other plans in all imaginable situations.

1.2.2 Invincible - Results

Invincible held his own against a varied set of opponents, winning three out of five games. It lost the first one comfortably because of the lack of experience. It won the next three games playing more aggressively, capitalizing on human unfamiliarity, and retaining its lead. It lacked the ability to detect the weakness of unusual approaches like the one used in the fifth game. It can never be told everything so there will always be situations that it can't handle correctly. The idea of using pre-defined plans worked to a degree but it will not be able to defeat high level opponents due to the logic behind pre-generated plans.

2 Nash Equilibrium Stratego Bots

Stratego is an example of a two-player zero-sum imperfect information game. Each player has their own policy π_i, which can be switched at any point during the game (episode). Stratego is a zero-sum game because the sum of the rewards between all players for a given action is 0. Since Stratego is a two-player game, this means that a player receiving a reward r means the other player receives a reward $-r$. A set of policies is in Nash equilibrium if no player can change their current policy to receive a higher reward. This effectively gives an ideal strategy for a given player in a game, so many Stratego bots are based on finding a Nash equilibrium.

2.1 ReBeL

ReBeL (Recursive Belief-Based Learning) is an algorithm created by Facebook's AI Research team in 2020, which is designed to converge to the Nash equilibrium of a two-player zero-sum game. ReBeL is an example of an RL+Search algorithm, which simulates intuition by searching the game tree to look ahead in the game and neural networks to make judgments on the value of states to make its next decision. Unlike previous RL+Search algorithms — such as AlphaZero — ReBeL is capable of handling imperfect information games. The ReBeL algorithm has been adapted to play a variant of Stratego called Stratego Tiny, which is played on a 4x4 board with 4 pieces per player. [4]

To handle imperfect information, we can model this information using **infostates** and **infosets**. An infostate for a player i is the set of states that cannot be distinguished with the information that player i has. An infoset for a player i is an infostate where it is player i's turn to make an action. Strategies

Table 2: Parameters used to train ReBeL [5]

Bot name	Description
d3_100	ReBeL trained on all the data, playing with CFR-depth 3
d3_50	ReBeL trained on the first 50% of the data, playing with CFR-depth 3
d3_25	ReBeL trained on the first 25% of the data, playing with CFR-depth 3
d2_100	ReBeL trained on all the data, playing with CFR-depth 2
cfr_d3	CFR with evaluation of non-terminal nodes to 0 and playing with depth 3
random	Chooses actions uniformly at random among the valid actions

Table 3: ReBeL tournament results [5]

P1/P2	d3_100	d3_50	d3_25	d2_100	cfr_d3	random
d3_100	-	98/154/4	122/129/5	190/45/21	136/120/0	237/19/0
d3_50	153/97/6	-	142/113/1	203/47/6	174/79/3	245/9/2
d3_25	187/69/0	154/101/1	-	203/51/2	182/74/0	252/2/2
d2_100	70/175/11	66/181/9	62/194/0 -	85/171/0	204/51/1	
cfr_d3	132/124/0	133/123/0	110/146/0	179/77/0	-	245/11/0
random	36/220/0	30/226/0	15/241/0	55/198/3	37/219/0	-

for both players are made with respect to their infosets. ReBeL also relies on a public belief state, which is a probability distribution over the histories (nodes in the game tree) that the game could be in. As more information is revealed, the public belief state is updated accordingly. [5]

2.1.1 Counterfactual Regret Minimization

Counterfactual Regret Minimization (CFR) is an algorithm that can solve the Nash equilibrium for an imperfect information game. However, it is too slow to use on its own as it requires searching the entire game tree to find the Nash equilibrium. For a game like Stratego, with a game tree of size 10^{535}, it's impractical to use the algorithm alone. Instead, it is used for the search part of RL+Search, up to a given depth of the game-tree. [5]

CFR is built on the idea of regret, which quantifies the desire to have taken another action instead of the current action that was taken. Positive regret means that the other action is prefered over the current action. Formally, the **counterfactual regret** for not taking an action a at infoset I_i under policy π, is defined as

$$r_i^\pi(I_i)(a) = v_i^{\pi_{I_i \to a}}(I_i) - v_i^\pi(I_i)$$

Additionally, we define the **counterfactual regret-sum** for action a at iteration t, for infoset I_i under policy π as

$$R_i^t(I_i)(a) = \sum_{\tau=1}^{t} r_i^{\pi^\tau}(I_i)(a)$$

Running over t iterations, the algorithm computes the regret-sum for a player i, then defines a new policy where actions with only positive regret are taken, weighted based on regret. After the t iterations are complete, the average of the policies is taken. This averaged policy approaches Nash equilibrium as $t \to \infty$.

2.1.2 ReBeL itself

ReBeL is trained in belief representation, based on the public belief state. A subgame is constructed with the root at the current state of the game. This game is solved to a certain depth using CFR. The value of each leaf in the game tree is estimated based on the infostate and the belief state with respect to the current policy as input to a neural network. ReBeL was trained with no outside input, relying on self-play to generate the values for each action. Various versions of ReBeL were trained based on CFR depth and percentage of information kept during training. (See Table 2)

There are 3 things to note from the results (See Table 3). The first is that the player to move first (Player 1) appears to have an advantage compared to other player, winning a larger percentage of

Table 4: Barrage P2SRO Results [6]

Name	P2SRO Win Rate vs. Bot
Asmodeus	81%
Celsius	70%
Vixen	69%
Celsius1.1	65%
All Bots Average	**71%**

games. The second is that larger CFR depths resulted in a higher win rate. This makes sense since this provides more information to make a decision. The third is that more data kept during training did not result in a higher win rate. This was unexpected for the author, and they posit that the reason behind this is that the data was somehow overfitting, leading to a more inaccurate network.

One issue with ReBeL is that it doesn't scale well. As the game tree grows, running CFR search becomes more costly, to the point of infeasibility. Since Stratego Tiny is smaller than the actual Stratego, it's unlikely that this method will be as effective when applied to Stratego.

2.2 Pipeline Policy Space Response Oracles

Policy Space Response Oracles (PSRO) are a class of algorithms that find an approximate Nash equilibrium for normal-form games. A **normal-form game** is a tuple (Π, U) where Π is the set of strategies for each player and U is the table of payoffs (rewards) for each player's policy. **Extensive-form games** contain more information about the game, such as turn-taking. Stratego is an example of an extensive-form game. Extensive-form games can be reduced to normal-form games for the purpose of solving Nash equilibrium. In theory, this would be bad, since turning an extensive-form game into a normal-form game increases the policies exponentially. In practice, this doesn't make a big difference in performance. [6]

2.2.1 Double Oracle Algorithm

The Double Oracle algorithm works by keeping a subset of policies at a given iteration, then finding a Nash equilibrium of those policies. Once that's been done, a policy is computed for each player to exploit this Nash equilibrium and is added to the set of policies being analyzed. PSRO approximates the Double Oracle algorithm by having each policy play each other and keeping track of the utility in a matrix. Then, an RL algorithm is used to compute a response to the Nash equilibrium. The issue with this algorithm is the sequential nature of it; only one policy is being computed at a given time. This is what the current paper addresses.

2.2.2 Pipeline PSRO

Pipeline PSRO (P2SRO) is a variant of PSRO that computes multiple policies at once in parallel, increasing the speed at which an approximate Nash equilibrium is reached. P2SRO splits the policies used into two sets: **fixed** and **active**. Fixed policies aren't being actively trained anymore but are still part of the set of policies. Active policies are being trained on all the policies lower than them in the pipeline. Once the lowest active policy reaches a training threshold, it becomes fixed, and a new active policy is added to the end of the pipeline. This algorithm lets multiple policies be trained at once and gives new active policies a head start, since they are trained on lower active policies in the pipeline.

An implementation of P2SRO was created with fictitious self-play used to calculate the Nash equilibrium of the current policies and the discrete-action version of Soft Actor-Critic used to find the best response policy. This implementation played Barrage Stratego, which is played on the same 10x10 board as regular Stratego, but with 8 pieces instead of 40. It was tested against various known Stratego bots that can play Barrage Stratego, including some that won past Computer Stratego Championships. After 820,000 episodes, the implementation had a win rate of around 71% against other bots (See Table 4).

Table 5: DeepNash Results [7]

Opponent	Number of Games	Wins	Draws	Losses
Probe	30	100.0%	0.0%	0.0%
Master of the Flag	30	100.0%	0.0%	0.0%
Demon of Ignorance	800	97.1%	1.8%	1.1%
Asmodeus	800	99.7%	0.0%	0.3%
Celsius	800	98.2%	0.0%	1.8%
Celsius1.1	800	97.9%	0.0%	2.1%
PeternLewis	800	99.9%	0.0%	0.1%
Vixen	800	100.0%	0.0%	0.0%

2.3 DeepNash

DeepNash is a bot created by DeepMind (a subsidiary of Alphabet) designed to play imperfect information games such as Stratego. DeepNash relies on an algorithm called Regularized Nash Dynamics (R-NaD) and a deep neural network to achieve an approximate Nash equilibrium. This approach is model-free, meaning the beliefs of the opponent aren't modelled when finding the Nash equilibrium. [7]

2.3.1 Regularized Nash Dynamics

R-NaD is a game-theoretic algorithm that evolves a policy with high fitness in a dynamic environment that allows it to converge to a Nash equilibrium. R-NaD is comprised of three steps: reward transformation, dynamics, and update. **Reward transformation** modifies the rewards of the game using a regularization policy.

$$r^i(\pi^i, \pi^{-i}, a^i, a^{-i}) = r^i(a^i, a^{-i}) - \eta \log(\frac{\pi^i(a^i)}{\pi^i_{reg}(a^i)}) + \eta \log(\frac{\pi^{-i}(a^{-i})}{\pi^{-i}_{reg}(a^{-i})})$$

The regularization constant η affects how fast the dynamics converge to a fixed point but increases the number of iterations (fixed points) needed to approach a Nash equilibrium. The **dynamics** step evolves the system based on replicator dynamics (which is similar to a regret minimization algorithm known as Follow the Regularized Leader) and is defined as

$$\frac{d}{d\tau}\pi^i_\tau(a^i) = \pi^i_\tau(a^i)[Q^i_{\pi_\tau}(a^i) - \sum_{b^i} \pi^i_\tau(b^i) Q^i_{\pi_\tau}(b^i)]$$

This dynamics system has a fixed point that it converges to, which represents a policy. This policy is used in the **update** step, where it becomes the next regularization policy, and the next iteration occurs.

2.3.2 DeepNash itself

DeepNash implements R-NaD using deep learning. The dynamics step of R-NaD is implemented in two parts: using a v-trace estimator to estimate the value function for the system, and Neural Replicator Dynamics (NeuRD); a policy gradient algorithm that modifies the softmax policy gradient to find the fixed point in the dynamic system. [8] The implementation was also fine-tuned at the end, to avoid annoying sequences of actions (e.g., moving one piece back and forth) for human players, as well as to avoid actions with sufficiently low probabilities.

DeepNash was tested on both humans and bots in the full Stratego game. DeepNash played 50 ranked matches on the Gravon website, winning 42 of them, and placing 3rd overall on the website's leaderboard. Against bots, DeepNash played up to 800 games against various leading Stratego bots and won over 95% of them (See Table 5).

3 Conclusion

When comparing the data-driven approaches to the Nash equilibrium approaches for Stratego, the Nash equilibrium approaches appear to outperform the data-driven approaches. This is likely due to

the nature of imperfect information games. The lack of information causes bluffing and deception to be a core part of the games, requiring some amount of novelty in play. Rigid approaches defined by data do not have the flexibility needed to perform these tasks at a high level for a game as complex as Stratego, whereas the Nash equilibrium approaches are formed using self-play, allowing for the necessary flexibility to arise naturally.

Among the Nash equilibrium approaches ReBeL is unable to scale for larger problems, which makes it an issue for practical applications. Between P2SRO and DeepNash, while they haven't played against each other, they have played against similar bots, and in that respect, DeepNash comes out on top. However, this may be due to DeepMind, the creators of DeepNash, having more funding and computing power compared to the researchers who made P2SRO. For further research, comparing P2SRO against DeepNash seems like a good approach. Getting some human tests against P2SRO would also help make a fairer comparison. Lastly, it would be interesting to see how DeepNash fares against top players in the Stratego World Championship, to see if it can surpass the top human players.

References

[1] Stankiewicz, J. A., & Schadd, M. P. (2009). Opponent modeling in stratego. *Natural Computing*.

[2] de Boer, V., Rothkrantz, L. J., & Wiggers, P. (2008). Invincible-A Stratego Bot. *International Journal of Intelligent Games & Simulation*, 5(1).

[3] Redeca, S., & Groza, A. (2018, September). Designing agents for the Stratego game. In *2018 IEEE 14th International Conference on Intelligent Computer Communication and Processing (ICCP)* (pp. 97-104). IEEE.

[4] Brown, N., Bakhtin, A., Lerer, A., & Gong, Q. (2020). Combining deep reinforcement learning and search for imperfect-information games. *Advances in Neural Information Processing Systems*, 33, 17057-17069.

[5] Falk, A. (2021). Stratego Using Deep Reinforcement Learning and Search.

[6] McAleer, S., Lanier, J. B., Fox, R., & Baldi, P. (2020). Pipeline psro: A scalable approach for finding approximate nash equilibria in large games. *Advances in neural information processing systems*, 33, 20238-20248.

[7] Perolat, J., De Vylder, B., Hennes, D., Tarassov, E., Strub, F., de Boer, V., ... & Tuyls, K. (2022). Mastering the game of Stratego with model-free multiagent reinforcement learning. *Science*, 378(6623), 990-996.

[8] Hennes, D., Morrill, D., Omidshafiei, S., Munos, R., Perolat, J., Lanctot, M., ... & Tuyls, K. (2019). Neural replicator dynamics. *arXiv preprint arXiv*:1906.00190.

Appendix

1. Advantage: "Unlimited" search depth: The search depth is unlimited because the depth is defined by the useful depth to detect and complete a plan. Because of the directional rather than exhaustive search this depth can easily be reached. This is a relatively small search problem that can easily be performed thousands of times without noticeable delay on modern hardware

2. Advantage: Possibilities to bluff: Bluffing can be implemented by defining what makes moves good for bluffing. Usually this means performing a normal plan with the wrong pieces. If "Invincible" can find a piece of which the opponent may believe it is the right piece for a certain plan and he uses it for that plan, then the opponent's belief that this piece has a rank when in reality it doesn't have is strengthened. This behavior is easy to implement when moves are chosen based on plans but very hard/impossible to include in a system that finds all the effects of a limited sequence of moves.

3. Advantage: Possibilities to coordinate attacks with multiple pieces: There are not many things in Stratego that a single piece can do, but plans that require multiple pieces generally also require many more moves. An exhaustive search of 12 plies allows to find plans that require 1 piece to move 6 fields, or 2 pieces to move 3 fields. That means both pieces must be already very close to where they are needed before a combination that requires these pieces can be found. If this is defined as a plan, then this plan can find the location of the required pieces and guide them where they have to be also as a search over a 10x10 graph.

4. Advantage: Focus computation on interesting situations: The computational is used more specifically for computing interesting aspects of a board position rather than for computing all the possible move sequences, out of which most are not going to be interesting

5. Disadvantage: "Invincible" knows only what it is told: Because there is no exhaustive search, everything that is not formulated as a plan will be completely ignored. If there is no plan telling "Invincible" to move a known low piece away when a known higher piece is approaching it then it will not do so. There is however only a limited number of useful things to pay attention to on short-term planning, and since the long-term plans can't be seen by exhaustive search anyway this is only a disadvantage in terms of programming time and not in playing strength
6. Disadvantage: Comparable information is calculated multiple times: Many plans use similar information, but can't use each others results if the calculation is split in independent sub-problems. This means that sometimes the same calculation is done multiple times. This is of course inefficient, but because time isn't really an issue this doesn't prove to be a big problem
7. Disadvantage: Game specific knowledge needed: So much of the search can be ignored only because of knowledge specific to the domain (in this case the game Stratego). A human expert in the domain will have to decide which plans should be added. In either case, a lot of the development time will go into formalizing human experience into computer-readable algorithms