Using RL to beat a Viking Board Game

Hadrien Helfgott
Department of Computer Science
McGill University
Montreal, QC H3A 0G4
`hadrien.helfgott@mail.mcgill.ca`

Abstract

The present paper evaluates the performance of various Reinforcement Learning approaches to Brandubh, an old Irish strategy board game. As a preliminary work, the project involved coding the rules of Brandubh and building a database of almost 700 000 boards using Monte-Carlo algorithms. Then, using the database, I trained 3 offline RL algorithms using different mechanics and evaluated their performance against both AI agents and humans.

1 Introduction & Background

Board games are a particularly fascinating topic in Reinforcement learning: their use of complex strategies coupled with typically human intuitions makes them an ideal playground to discover how far can algorithms truly *think*. In this project, I focus on Brandubh, an old Irish Viking siege game. Exploring this game should be particularly interesting. First, to my knowledge, this game has never been tackled by Reinforcement Learning before. My project will therefore give some insight into the development of RL board game algorithms without previous human or algorithmic expertise. Then, like chess, Brandubh is way too complex to be solved : its state-space complexity is estimated to 10^{14} (Compy et al., 2021). Finally, unlike chess and most western board games, Brandubh is an inherently asymmetrical game. The two players, the defender and the attacker, have radically different goals (see Appendix 1 for precise Brandubh rules), therefore they will play very differently (see Appendix 2 for several sample games). This means that an agent can be a performant attacker, but a very weak defender. Not only are the styles different, but the odds are asymmetrical too: both online sources and my own experiments confirmed that it is significantly easier for a defender to win than for an attacker (roughly 70 to 30%).

2 Methodology

I first attempted to do online TD learning using typical Q-learning mechanics for function approximation of state values. This attempt was unsuccessful, and the algorithm did not converge after tens of hours of self-playing. Although a purely self-playing approach may ultimately be able to perform at Brandubh, the resources and time at my availability made it risky to follow on that way despite weak first results. I therefore decided to orient myself towards offline RL.

2.1 Building a database

I decided instead to create a massive database of games that I would use to develop my offline RL algorithms. These algorithms would iterate through all the possible moves at each step and choose the best one using an evaluation function learned using the games database. This meant that the quality of this database, if too low, would severely hinder the potential of my algorithms. Therefore, I developed two non-RL agents to play Brandubh with human-like ability and fill the database:

Odin: This first agent is a MCTS algorithm that uses a simple heuristic (Appendix 5a) to sort moves from best to worst. It then picks the k best moves, uses heuristic-based agents (called *Hugin* and *Munin*) to play n games following UCT formula for exploration, and returns the move which most consistently results in victory. The Odin I used for the database had settings k=5 and n=20 and was strong enough to play at a low human-level.

Leif: This agent was similar to Odin but chose 20% of its moves randomly: its purpose was to bring more diversity in the database and to give future RL agents the ability to see bad moves being played and sanctioned. For the database I used mostly k=3 and n=10, but also 'enhanced' settings of k=6 and n=25.

I recorded games in board bins, with one board state per row. Each board was associated to the final return of the corresponding game. Return was computed using only the outcome of the game: a reward of +10 for a defender victory, -10 for an attacker victory, 0 for a draw — with a discount factor of 0.99. This meant that two boards at a very different stage of the game would both be associated with the *same* final return. Although this may seem like a risk given the resulting disparity of boards with similar returns, the sorting of bins by king position (see below) avoids much of the issue. After a week of board collection, (Appendix 3) I had to start writing and testing the different algorithms. At that stage, I had obtained a bin of 679 700 board positions coming from different player's games.

2.2 Board treatment

All boards in the database were processed before being integrated by players:

Rotation: Brandubh boards are both vertically and horizontally symmetrical. This means that a 90° rotation of a board leads to an absolutely equivalent situation. Therefore, I rotated all the boards so that the king would be in one of the upper-left squares — as result, the variance of the database was significantly reduced.

Sorting: Not all board positions can be analyzed with the same factors — all of my attempts at training agents on the full bin at once failed. Instead, I chose to split the database it in 26 smaller bins corresponding to the 13 possible king positions left, and on whether it was the attacker or the defender's turn. Therefore, a movement would only be analyzed in regard to a database with all the same-player positions where the king was in the same spot. This meant training models independently for all 26 bins (see Appendix 4 for more details).

In each agent, a similar rotation and sorting method was used to evaluate every new board — in order to analyze them under the same conditions as the database.

2.3 RL Agents

I programmed 3 agents who all selected their moves by computing the expected final return of a game starting from each potential next position, and then picking the best position according to this metric (looking for the highest reward if defender, for the lowest reward if attacker). The agents were built using different methods:

Loki is a simple function approximation agent. Each board corresponding to a possible next move is converted to an 11-long feature vector, mostly comprised of high-level features (e.g. number of enemies that can access the king in one move; see Appendix 5c). It then predicts the outcome (final return) of the game. Loki uses one different Linear Regression model (see Appendix 6 for hyperparameters) for each of the 26 bins and each model was trained with supervised learning using the database.

Heimdall is a less huaman-guided agent. Although it does implement a few high-level features, its 157-long feature vector (see Appendix 5b) is mostly made of raw board information encoded in different ways, with additional piece-by-piece information such as how far in each direction a piece can move. The feature building for this board was heavily inspired by chess TD-learning algorithms (Lai, 2015). Heimdall uses deep MLPs to compute the expected outcome of a board, each MLP having 3 linear activation hidden layers (see Appendix 6 for hyperparameters).

Thor is a simple bootstrapping agent that does not train some set of weights. When facing a new board, Thor computes the similarity of this board with all the other boards in the corresponding bin using a pre-designed filter system (see Appendix 7). It picks the n most resemblant boards and computes the expected value of the boards by simply averaging over the actual final reward of these resemblant positions. Therefore, Thor does not need training per se, only to be given more and more positions to analyze

2.4 Learning curves and Evaluation

In order to evaluate the performance of my agents, I shuffled the board list and trained each agent on an exponentially growing number of positions to obtain a full learning curve with defense and attack performances against my MCTS algorithm Odin (see Appendix 8 for details). Then, to assess the performance of all of my agent against a human player, I played 128 games against my 3 fully-trained RL agents and Odin. Both the agents and the player status (attack/defense) were randomized so my own learning curve would not affect some agents more than others. For context, I am not a long-time player of Brandubh but had played around 100 games during the last month for testing and investigating purposes before the start of these trials. I also have some board game experience as an intermediate level chess player. To evaluate performance, I recorded the victory rate of the agents, game lengths, and my own average decision time for moves against each opponent, expecting tougher opponents to make me think longer before I play. Draws were declared after 60 moves without a definite winner.

3 Results

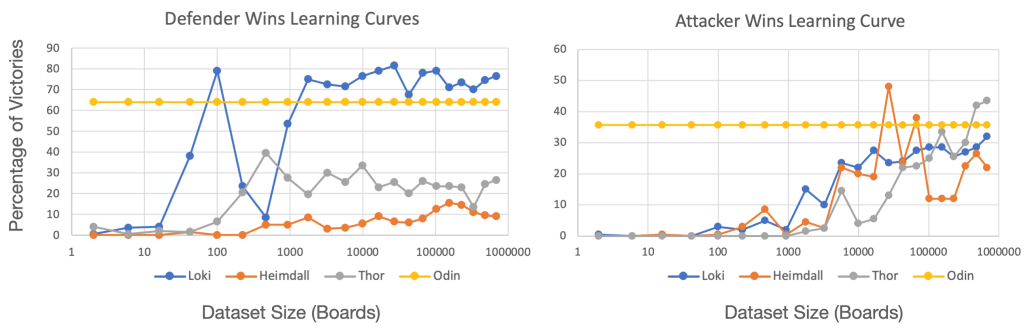

Figure 1: Learning curves of RL algorithms in attack and defense, using win rate over database size

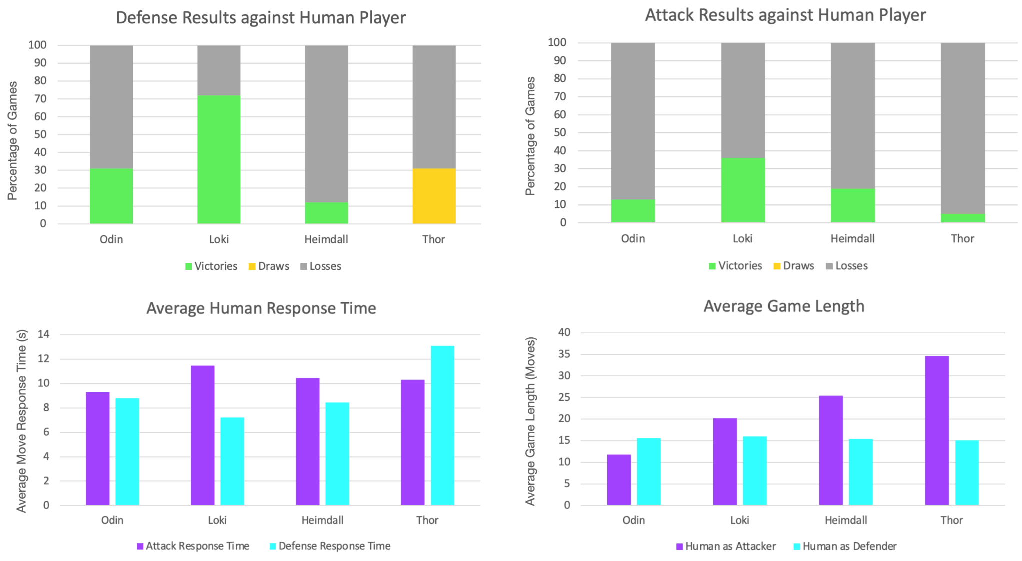

Figure 2: Plots of RL algorithms performance against a human player in win rates, *human* response time and game length

4 Discussion

All three algorithms seem to show very different performances despite being trained on the same dataset (see Appendix 9 for detailed results). The high-order Linear Regression algorithm Loki slightly outperforms Odin (MCTS) in defense, and almost reaches its level in attack. However, it seems to be particularly efficient against a human player, winning most of its games as defender and almost 40% as an attacker. The MLP agent Heimdall using low-level features has very low performance in defense against both Odin and a human player. The only area where it does significant progress is in attack against Odin — although the peak around 20 000 in its learning curve seems to be due to luck, as I was not able to replicate it over targeted trials. With poor performance against Odin in defense, the bootstrapping algorithm Thor is however the only algorithm that outperforms it in attack. However, this performance crumbles against a human player: it does not manage to get any victories in defense, only to stale for draws. In attack, it is also by far the weakest agent.

It should be noticed that it took longer to agents to learn to attack than to defend. This is likely due to the fact that king position, the only determinant factor for defender victory, is at the center of the bin division — while the many different ways to capture a king all involve multiple pieces and specific positions that take more time to be learned.

Human response times seem to indicate Loki as the most challenging agent in defense, and Thor as the most challenging in attack. However, this data should be taken carefully: as I was playing the game, I noticed that my reaction times were less modulated by the difficulty of the opponent than by its unpredictability — even if this unpredictability led to bad moves. Game lengths show a similar pattern: they are very similar when the algorithms are attackers, but in defense it mostly highlights how unprepared, low-performing agents stale for time instead of winning games.

The main takeaway from this experiment is the likely the importance of human knowledge in Brandubh RL, at least at this early developmental stage. Loki managed to achieve victories because the high-order features it was coded with already gave much human guidance — Heimdall, with low-order features, had much worse performance despite having a much more complex neural network. Thor, the most atypical agent, was also the worse one by far: it performed decently against Odin, the agent its database was made with, but as soon as a different opponent appeared it fully crumbled — not having pre-existing, feature-guided game knowledge made it totally inefficient. It should also be kept in mind that all three algorithms only reached their level of performance because bins were divided based on king position — indicating king position as the most important factor of a board is already a very strong helper for these programs, that would have struggled much more without this information. Overall, I conclude a heavy human-assisted approach is still needed to rapidly develop a Brandubh RL agent — at least when lacking much more computational resources.

5 Future Directions

Future improvements to this approach would likely include the following:

Enlargement of the database. The database I used consisted of 700 000 board positions (equivalent to roughly 55 000 games), but as shown in the Results section, learning curves rarely reached a true plateau by the end of the training. Having more time to build a bigger dataset would likely increase overall performance.

Enrichement of the database. At least as important as database size, database diversity should be improved. The gap in Thor's performance against the MCTS agent and against a human clearly showed the non-generalizability of my database: being also filled with human games would likely help the algorithms.

More frequent rewards. I only rewarded algorithms once, at the end of the game, depending on their result. Adding rewards when the algorithms eat an enemy piece for example could help algorithms to adopt less king-focused strategies.

Add Monte-Carlo mechanics to RL agents. A successful approach could consist in anticipating games way past the next move, and then running ML-trained evaluation functions on an array of possible future boards instead of only doing so on the next position. This could lead to the real-time elaboration of more complex strategies such as traps or baits.

6 References

[1] Compy, K., Evey, A., McCullough, H., Allen, L., & Crandall, A. S. (2021) An Upper Bound on the State-Space Complexity of Brandubh arXiv. https://doi.org/10.48550/ARXIV.2106.05353

[2] Lai, M. (2015) Giraffe: Using Deep Reinforcement Learning to Play Chess (Version 2). arXiv. https://doi.org/10.48550/ARXIV.1509.01549