BeeMo, a Monte Carlo Simulation Agent for Playing Parameterized Poker Squares

Karo Castro-Wunsch, William Maga, Calin Anton
MacEwan University, Edmonton, Alberta, Canada
karoantonio@gmail.com, magaw@mymacewan.ca, antonc@macewan.ca

Abstract

We investigated Parameterized Poker Squares to approximate an optimal game playing agent. We organized our inquiry along three dimensions: partial hand representation, search algorithms, and partial hand utility learning. For each dimension we implemented and evaluated several designs, among which we selected the best strategies to use for BeeMo, our final product. BeeMo uses a parallel flat Monte-Carlo search. The search is guided by a heuristic based on hand patterns utilities, which are learned through an iterative improvement method involving Monte-Carlo simulations and optimized greedy search.

Introduction

This paper describes the design of BeeMo, a Parameterized Poker Squares agent we developed to participate in the EAAI-2016 student-faculty research challenge. We present an overview of our inquiry as well as the most important aspects of algorithm and agent design: implementation of flat Monte Carlo search, state space representation, and use of Monte Carlo simulations for feedback learning. To facilitate the reproducibility of our results and to encourage further work on Parameterized Poker Squares, we made the Java source code of the agent available at: https://github.com/MrMagaw/Beemo.

Parameterized Poker Squares

The 2016 EAAI NSG Challenge asks participants to submit agents to play Parameterized Poker Squares (Neller 2014). Poker Squares, a variation of Solitaire, is played by placing cards in a 5×5 grid, where each row and column is a poker hand. The aim of the game is to maximize the total points of the ten hands, according to a given scoring policy, usually the American or the British system. Poker Squares is a stochastic single player game, with a very large search tree size.

Parameterized Poker Squares is a variation of the game in which the scoring system is non standard and revealed right at the beginning of the game. The lack of initial knowledge about the scoring system and the wide range of possible scoring systems have important consequences for the agent design. For example, the American scoring system, which values royal flushes highly and single pairs minimally, necessitates the placement of flushes in either columns or rows exclusively. However, a scoring system that rewards only high cards and penalizes all hands requires the avoidance of any hand. This forces agents to use more generalized algorithms and approaches and prevents the implementation of simple heuristic based agents.

The guidelines of the competition specify a limited training time for the agent to learn the scoring system and a limited play time per game.

Agent design

We identified three axes along which to design our Parameterized Poker Square agent: hand representations, search algorithms, and partial hand utility learning algorithms. Initially we investigated the choices for each axis independently, without considering possible inter axes interactions. After selecting the best approaches along each axis we investigated the performance of complete agents by combining the selected approaches and comparing their overall score.

Hand Pattern and Board Context Encoding

We used a straightforward representation of the board, enhanced with a bit packed representation of the remaining deck of cards which allows fast search and comparison.

The simplistic representation of a hand as a list of cards is undesirable due to its size and inconsistency: similar hands (for example one pair) have many different representations. Therefore, we implemented a new hand encoding scheme based on the concept of hand patterns. Each partial hand was reduced to an abstract representation, referred to as a hand pattern, which retained only the important information. The patterns are based on the following principles:

- The order of the cards is irrelevant
- The rank of the cards is irrelevant, with the only exception being the possibility of pairs, three or four of a kind and straights
- The suit of the cards is irrelevant as long as enough information is retained to determine whether a flush is possible.

Therefore, the information used to represent a pattern is:

- If it has a flush – 1 bit
- If it has a straight – 1 bit
- The number of cards without a pair – 3 bits
- Number of pairs – 2 bits
- If it has three of a kind - 1bit
- If it has four of a kind – 1bit
- If it is a row or a column – 1 bit.

Notice that our hand pattern encoding does not identify a royal flush. We chose not to include this information because our experiments indicated that the marginal performance increase it creates is significantly outweighed by the additional computational time. This is due to the extreme rarity of royal flushes.

This hand pattern encoding is too coarse as it misses context information. For example, a three of a kind with the possibility of becoming a four of a kind is encoded exactly the same as a three of a kind with no possibility of becoming a four of a kind. These two hands have different values and should be identified accordingly. We extended the basic pattern encoding to incorporate the following information about the cards left in the deck:

- Number of cards of the primary rank – 2 bits
- Number of cards of the secondary rank – 2 bits
- Number of cards that can lead the hand to be a flush – 2 bits representing: 0 if there are not enough cards left of the relevant suit to complete a flush, 1 if there are just enough cards left to complete a flush, and 2 if there is an excess of cards left to complete a flush.

The primary rank is the rank with the most number of repetitions in a hand and the secondary rank is the rank with the second highest number of repetitions. Hands in which there are more than 3 ranks, and therefore have no possibility of becoming a full house or four of a kind, have the number of cards in the secondary rank set to 0 to reduce complexity. Empirical evidence shows that the contextual information increases the number of unique patterns from 60 to around 550. Our experiments with several search algorithms and learning approaches indicate that the added contextual information leads to more consistent training and results in better overall performance.

The hand pattern uses only 16 bits but, for simplicity, we represented it as a 32 bit integer in our final Java implementation.

Search Algorithms

We developed several classes of search algorithms: simple rule based, expectimax, optimized greedy (OG), and Monte Carlo variations (Brown et al. 2012). For comparing the algorithms we employed a set of hand pattern utilities derived from our own experience with the game of poker under the American scoring system.

Rule Based Search

It uses if-else statements in order to direct the search along a specific game play path and/or strategy. The strategy employed was custom suited to the 'expert knowledge' we could extract from our own experience playing the game on the American scoring system. Despite being very fast, the agent is extremely rigid and non-adaptable to new scoring systems. The algorithm scored an average of less than 100 points taking almost 1 second to complete 10,000 games. This algorithm was also used to guide Monte Carlo simulations.

Expectimax Tree Search

Our implementation of the algorithm evaluates all nodes at a fixed depth and back propagates the values up to the top node. When a chance node is encountered, values of all child nodes are averaged and the result is propagated up. When a choice node is encountered, the value of the best child is propagated up. This approach achieved a score of 119.1, at a search depth of 2, taking 30 minutes to play 10,000 games.

Optimized Greedy

This is a simple greedy algorithm that uses hand pattern values to evaluate, compare and choose play positions. Any new card taken from the deck can be placed in up to 25 positions. The value of each position depends on only two hands, one row and one column, which intersect at that position. The contribution of each of these hands to the position's value is the difference between the utilities of the hands after and before placing the card. The sum of these two differences is used to express the value of each position. The new card is placed at the highest valued position. The pattern utilities are stored in a hash, so the main computational load of the algorithm is the encoding of the hands into patterns, which is done efficiently through bit manipulations. The algorithm plays tens of thousands games per second, and scores an average of 119 points on 10,000 games. Its speed and adaptability to different scoring systems make it an excellent choice for guiding Monte Carlo simulations.

Monte Carlo

Monte Carlo techniques have been proved useful for stochastic, large state-space problems like Poker Squares. They produce a fairly accurate model of the state-space without analyzing every single node. Thus, we used Monte Carlo methods for searching as well as learning hand pattern utilities.

Flat Monte Carlo

The simplest form of Monte Carlo search we implemented was modeled after the Imperfect Information Monte Carlo algorithm (Furtak and Buro 2013). The algorithm explores a given choice node by playing a large number of simulated games from each of its children, estimating the value of a child as the average score of the corresponding simulations, and selecting the move which results in the highest value child. The flat characteristic of the algorithm stems from the uniform distribution of games among child nodes. Any of the algorithms described above can be used to guide the simulated games provided they are reasonably fast (expectimax is thus unsuitable). We implemented flat MC search with a random player, a rule based search, and an optimized greedy search. While fast, the random player based variant had poor performances; half the score of the other approaches. The rule based search MC and the optimized greedy based MC had comparable performances, but different time requirements (see Table i).

UCT Monte Carlo

One of the most powerful and widely implemented improvements of the classic Monte Carlo approach is the Upper Confidence Threshold (UCT) (Kocsis and Szepesvari 2006). It alters the uniform distribution of node selection by trying to balance the exploitation of nodes with high average score and the exploration of nodes with low average score. The flat Monte Carlo algorithm described above is, in this respect, entirely exploratory as it visits each node the same number of times. For example, suppose that after 400 simulations for each node, one of them has an average score of 50 while all the others have scores above 100. The low score node is an obvious non-contender. Despite this information, flat MC search will distribute the remaining simulations uniformly among all nodes. UCT allows the search algorithm to focus on higher score nodes without completely disregarding lower score ones. However, our own implementations of UCT turned out to be more problematic than expected as they always resulted in lower scores than flat MC. Moreover, evaluating all nodes before every simulation (to decide which one to explore) significantly increased the running time and complicated the parallelization of simulations.

BeeMo

For our final implementation we decided to use the flat Monte Carlo search with simulated games guided by the optimized greedy search. The main factors in our decision were: superior performance, reasonable running time and flexibility of adapting the algorithm to other scoring systems. Additionally, BeeMo's speed was significantly improved through parallelization which was deeply integrated due to the simple nature of flat MC search. BeeMo takes full advantage of multi core processors, by creating several threads for game simulations, which are then run in parallel on the available cores. In Table i, we present a comparison of average scores of some algorithm implementations using the American score system on a test batch of 10,000 games. For MC search variations 100 games were simulated at each node.

Algorithm	**Average Score**	**Running time for 10,000 games**
Rule Based	<100	~1 second
Expectimax	117	30 minutes
OG	119	~ 1 second
Flat MC + Rule Based	123	~ 7 hours
Flat MC + OG	125.3	~ 8 hours
BeeMo (Parallel Flat MC + OG)	125.3	~ 4 hours

Table i. Relative performance of search algorithms.

Learning

Due to the unknown scoring scheme, learning the partial hands' utilities is highly important to the performance of the agent. The Monte Carlo simulation was used for learning, albeit from a slightly different angle: the simulations were employed to estimate the utility of each partial hand pattern. The learning algorithm uses rounds of 10,000 Monte Carlo simulations. Each round consists of a training phase followed by an evaluation phase.

In the training phase, 5,000 simulated games are played, through which the hand pattern utilities are produced. At the beginning of the training phase all pattern utilities are arbitrarily set to zero. In every simulated game, the utility of each partial hand is set to the corresponding final hand value. For example, if a partial hand consisting of only a 5H, results (at the end of the game) in a flush with a score of 20, its utility is set to 20 points. At the end of a simulated game, each pattern utility is computed as the average of the scores of complete hands that resulted from hands which matched the pattern. The updated set of pattern utilities is used to guide the greedy search in the next simulated game. This leads to a positive feedback loop which greatly improves the score of the agent. This feedback loop is the most potentially novel component of the agent. In Table ii, we present a comparison of average scores with and without feedback loop on tests batches of 10,000 games. With the only exception of custom scoring system, which alternates negative and positive scores, and has a large penalty for high cards, the feedback loop improves the average scores.

	Flat MC	UCB1
# patterns seen	350	352
American Score	120.2	121.5
British Score	47.8	47.8
Custom	160.1	153.0
High Card	83.1	89.9
Hyper Corner	-0.53	-0.63

Table ii. Average scores of training with and without feedback.

In the evaluation phase 5,000 games are played, using the set of hand pattern utilities produced during the training phase. The overall score of a set of hand pattern utilities is estimated as the average of the evaluation games' score. The set of pattern utilities with the maximum overall score is kept for the final playing agent.

Flat Monte Carlo training

Due to the advantages presented above, we chose flat Monte Carlo simulation as our first training approach. It created hand pattern utilities which resulted in average scores comparable to our hand crafted values. For the American score system parallel flat MC with OG guidance obtained an average score of 125 points when it used the crafted hand pattern utilities and an average score of 120 points when it used the flat MC learning. The running time was reduced from about 7 seconds to just below 1 second. The accuracy of this utility estimation scheme depends on the number and frequency of hands that are matched to a certain pattern. Some low frequency but possibly high utility patterns are only rarely updated, which may result in unreliable utility estimations.

Score System	Without Feedback	With Feedback
American	93.7	115.3
British	40.5	43.4
Custom	157.9	154.4
High Card	8.9	14.3
Hyper Corner	-0.97	-0.61

Table iii. Training results of Flat MC and UCB1 simulations.

Flat UCT training

To check the accuracy of these estimations we implemented a UCT evaluation scheme inspired by the UCB1 variation of the Upper Confidence Bound class (Auer, Cesa-Bianchi, and Fischer 2002.), with a small exploration term parameter (C_p), which we optimized empirically.

As shown in Table iii, UCB1 slightly increased the number of patterns seen. However its influence on performance was positive for some scoring systems and negative for others.

BeeMo's training

To benefit from both approaches, BeeMo uses both evaluations schemes; one in three simulated games uses UCB1. The interplay between alternating searching and training components produces an initially random-like behavior which becomes more robust as patterns' utility estimations become more accurate. As Figure 1 illustrates, BeeMo's learning algorithm belongs to the class of iterative improvement methods.

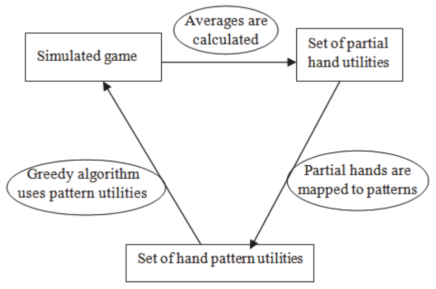
Simulated game
Averages are calculated
Set of partial hand utilities
Partial hands are mapped to patterns
Set of hand pattern utilities
Greedy algorithm uses pattern utilities

Figure 1. Pattern Utility Estimation Feedback Loop

Conclusions

We implemented an optimized greedy search algorithm guided by learned hand pattern utilities. Based on our own implementation and testing of various search algorithms, flat Monte Carlo search guided by optimized greedy was determined to be the most effective approach. For utility learning we used a combination of flat Monte Carlo simulations and modified UCB1 scheme which effectively improved the performance of the final agent.

References

Neller, T. 2014. NSG Challenge: Parameterized Poker Square, http://cs.gettysburg.edu/~tneller/games/pokersquares/eaai/paramPokerSquares.pdf

Browne, C.; Powley, E.; Whitehouse, D.; Lucas, S.; Cowling, P.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S. and Colton, S. 2012. A Survey of Monte-Carlo Tree Search Methods. *IEEE Trans. on Computational Intelligence and AI in Games*, 4(1):1-43.

Auer, P; Cesa-Bianchi, N. and Fischer, P. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.* 47(2):235–256.

Furtak, T. and Buro, M. 2013, Recursive Monte Carlo Search for Imperfect Information Games. *In Proceedings of the IEEE CIG*

Kocsis, L, and Szepesvari, C. 2006. Bandit based Monte Carlo Planning. *ECML-06. Number 4212 in LNCS*