Learning and Using Hand Abstraction Values for Parameterized Poker Squares

Todd W. Neller and **Colin M. Messinger** and **Zuozhi Yang**
Gettysburg College
tneller@gettysburg.edu

Abstract

We describe the experimental development of an AI player that adapts to different point systems for Parameterized Poker Squares. After introducing the game and research competition challenge, we describe our static board evaluation utilizing learned evaluations of abstract partial Poker hands. Next, we evaluate various time management strategies and search algorithms. Finally, we show experimentally which of our design decisions most significantly accounted for observed performance.

Introduction

The inaugural EAAI NSG Challenge[1] was to create AI to play a parameterized form of the game Poker Squares. Our best approach first applies Monte Carlo ϵ-greedy reinforcement learning to the task of learning a static (i.e. heuristic) evaluation function that estimates the expected final score for an abstraction of the game state. It then plays the game with expectimax search using this static evaluation at a shallow depth cutoff.

In this section, we describe the game of Poker Squares and the parameterization of the game. In the next section, we focus on different approaches to the static evaluation of nonterminal game states and experimental results comparing simple greedy play with such approaches. We then turn our attention to time management, i.e. the apportionment of search time during real-time play. Finally, we compare different search techniques, gaining insight to which work best for this problem domain.

Poker Squares

Poker Squares (BoardGameGeek.com 2015; Neller 2013a) (a.k.a. Poker Solitaire, Poker Square, Poker Patience) is a folk sequential placement optimization game (Neller 2013b) appearing in print as early as 1949, but likely having much earlier origins. Using a shuffled 52-card French deck, the rules of our oldest known reference (Morehead and Mott-Smith 1949, p. 106) read as follows:

> Turn up twenty-five cards from the stock, one by one, and place each to best advantage in a tableau of five rows of five cards each. The object is to make as high a total score as possible, in the ten Poker hands formed by the five rows and five columns. Two methods of scoring are prevalent, as follows:
>
> | HAND | ENGLISH | AMERICAN |
> |---|---|---|
> | Royal flush | 30 | 100 |
> | Straight flush | 30 | 75 |
> | Four of a kind | 16 | 50 |
> | Full house | 10 | 25 |
> | Flush | 5 | 20 |
> | Straight | 12 | 15 |
> | Three of a kind | 6 | 10 |
> | Two pairs | 3 | 5 |
> | One pair | 1 | 2 |
>
> The American system is based on the relative likelihood of the hands in regular Poker. The English system is based on the relative difficulty of forming the hands in Poker Solitaire.
>
> You may consider that you have "won the game" if you total 200 (American) or 70 (English).

For our purposes, players seek to commit randomly drawn cards sequentially to 5-by-5 grid positions so as to maximize the expected total score, i.e. the sum of the ten row and column hand scores according to the given point system. Note that the single remaining Poker hand classification of "high card" scores no points in the above point systems.

Parameterized Poker Squares

As David Parlett observed, "British scoring is based on the relative difficulty of forming the various combinations in this particular game, American on their relative ranking in the game of Poker." (Parlett 2008, pp. 552–553) We observe that different point systems give rise to different placement strategies.

For example, in playing with British or American scoring, one often has a row and column where one dumps unwanted cards so as to form higher scoring combinations in the other rows and columns. However, a large negative score (i.e. penalty) for the "high card" category would discourage leaving any such row or column without a high probability of alternative scoring.

In our parameterization of Poker Squares, we parameterize the score of each of the 10 hand categories as being an integer in the range $[-128, 127]$. Given a vector of 10 integers corresponding to the hand classification points ordered

[1]Whereas DARPA has its Grand Challenges, ours are Not-So-Grand.

from high card to royal flush as in the table above, the player then plays Poker Squares according to the given point system.

Contest point systems included American, British, Ameritish, random, hypercorner, and single hands. Ameritish systems are randomly generated with a royal flush maximum hand score in the range of the royal flush scores of both the British and American systems ($[30, 100]$). Other hands are randomly assigned a proportion of the generated maximum hand score that falls in the range of such proportions for both systems. Thus, Ameritish systems range from American to British in their maximum scores and proportions of lesser scores. Random point systems are randomly generated by the assignment of random scores in the range $[-128, 127]$ to each hand category. Hypercorner systems are randomly generated by the random assignment from the set $\{-1, 1\}$ to each hand category. Single hand systems score 1 point for a single hand category and 0 points for all other hands.

The contest goal is to design a Parameterized Poker Squares AI player that maximizes expected score performance across the distribution of possible point systems within real-time constraints.

Static Evaluation

For each point system tested in contest evaluation, each AI player is given the point system and 5 minutes to perform preprocessing before beginning game play. For each game, each player is given 30 seconds of total time for play decision-making. One hundred games are then played with the given point system, and players are rewarded a tournament score on a linear scale from 0 to 1 from the minimum to maximum total game points, respectively. Thus the 5 minutes should be well-utilized to create an evaluation function for the game state that is able to efficiently provide a good approximation of the expected score of any game state for any point system.

While the total state space is too large to evaluate beforehand, we take the computer Poker research approach of abstracting the state space and then applying on-policy Monte Carlo reinforcement learning in order to simultaneously improve estimates of the abstracted game and improve play policy that guides our Monte Carlo simulations.

Abstracting Independent Hands

Our Naïve Abstract Reinforcement Learning (NARL) player abstracts the state of each independent row/column and learns the expected value of these abstractions through Monte Carlo ϵ-greedy reinforcement learning. Each hand abstraction string consists of several features which we considered significant:

- Number of cards played in the game so far
- Indication of row ("`-`") or column ("`|`")
- Descending-sorted non-zero rank counts and how many cards are yet undealt in each of those ranks appended to each parenthetically
- Indication of whether or not a flush ("`f`") is achievable and how many undealt cards are of that suit
- Indication of whether or not a straight ("`s`") is achievable
- Indication of whether or not royal flush ("`r`") is achievable

For example, "`14|1(3)1(2)1(2)f(8)s`" represents a column hand abstraction after the 14th move. There is one card in each of three ranks, two of which have two of that rank undealt and one has three undealt. A flush is achievable with eight cards undealt in that suit. A straight is achievable and a royal flush is not.

During Monte Carlo reinforcement learning, such hand abstractions are generated and stored in a hash map. Each abstraction maps to the expected hand score and number of occurrences of the hand. These are constantly updated during learning. By storing the expected scores of each row/column complete/partial hand, the hash map allows us to sum scoring estimates for each row and column, providing a very fast estimate of the expected final score of the game grid as a whole. Note that this naïvely assumes the independence of the hand scoring estimates.

Raising Proportion of Exploration Plays

During the Monte Carlo reinforcement learning stage, we use an ϵ-greedy policy with a geometric decay applied to the ϵ parameter. Thus for most of time the player chooses an action that achieves a maximal expected score, but also makes random plays with probability ϵ.

In our initial application of ϵ-greedy play, $\epsilon = 0.1$ with geometric ϵ-decay $\delta = 0.999975$ per simulated game iteration. However, we empirically observed that if we significantly raise the initial value of ϵ to 0.5, increasing initial exploration, the player has a better performance.

In addition, the time cost for random play is much less than greedy play, so increasing the proportion of random plays increases the number of overall learning iterations per unit time. Empirically, this relatively higher ϵ will not only raise the number of exploration plays but also will be able to leave sufficient time for exploitation plays. Note that sufficient exploitation play is necessary for long term planning (e.g. royal flush, straight).

Considering Frequency of Partial Hand Sizes

We observed our player's behavior and found that it tended to spread cards evenly among rows and columns in the early and middle stages of the game. The reason for this behavior is that the player is making greedy plays that maximize expected score gain. In a pre-evaluation between NARL and another player developed earlier that performed better under the single-hand Two Pairs point system, we observed that with same card dealt, NARL tended to set up one pair evenly among rows and columns according to the assumption of hand independence, while the comparison player appeared to gain an advantage by preferring to focus on developing a row/column with a pair and two single cards early.

Based on this observation, we added the current distribution of hand sizes to the abstraction. The number of cards

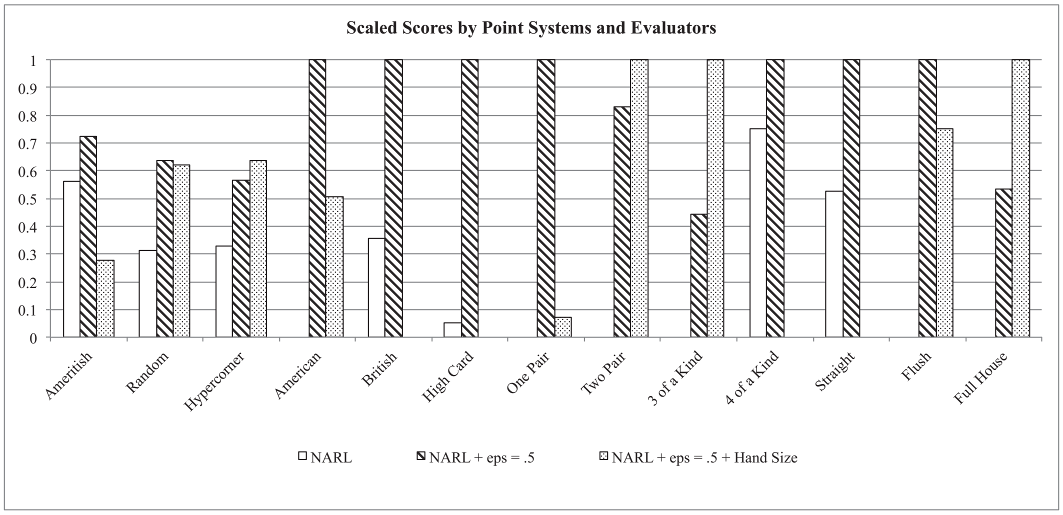

Figure 1: Comparison of learning evaluation performance. NARL with $\epsilon_0 = 0.5$ performed best for most point systems.

played in each row and column are tallied, and we summarize the distribution in a hand size frequency vector represented as a string. For instance, the string "`721000`" represents a grid hand size distribution after the 2nd move. (The number of cards dealt can be inferred from the abstraction.) The zero-based index of the string corresponds to hand size in a row/column. Thus, "`721000`" indicates that there are seven empty hands, two with one card, one with two cards, and none with more than two.

The previous grid hand abstraction is trained together with hand size abstraction to learn the difference between the final score and expected score at each of the 25 states across the game. In practice, we find that adding this abstraction feature generally improves performance for some simpler point systems

Experiments and Data

We experimented with 3 players, all of which used ϵ-decay $\delta = 0.999975$. The first used an initial epsilon $\epsilon_0 = 0.1$, whereas the second and third used $\epsilon_0 = 0.5$. Only the third player incorporated the hand size frequency abstraction feature.

For each random point system (the Ameritish point system, Random point system, Hypercorner point system) we generated a sample of 500 systems and measured the average greedy-play performance of 2000 games for each player and system. For fixed point systems, we collected average performance of 2000 games for each player and system. For each point system, performance was scaled between 0 and 1 as with the previously described tournament scoring.

Results are given in Figure 1. NARL with $\epsilon_0 = 0.5$ dominated all but three point systems where the addition of the hand size abstraction proved best.

Real-Time Search: Time Management and Search Algorithm

Paths through the Poker Squares game tree alternate between chance and choice nodes. Given unbounded computational time, one could perform a depth-first expectimax evaluation of the game-tree in order to inform perfect play decisions. However, the size of the game-tree prohibits such an approach.

There are two top level design decisions in our approach. First, we choose a time management policy that focuses search effort during the points of the game where it is of greatest benefit to play. Next, we choose a search algorithm that best utilizes our approximate static evaluation in order to discern nuances lost in our abstraction and more closely approximate optimal play.

Time Management

The time given to make play decisions should be distributed such that score is maximized. To manage time more efficiently, all time management policies give no consideration to the first and final plays, as these are inconsequential and forced, respectively. Two time-management system we used to distribute allotted time across turns were:

- *Uniform time management* (**UTM**) divides the given time evenly between all plays that were to be considered.
- *Gaussian time management*, **GTM**, distributes time according to a Gaussian curve as in the "MID" time-management policy of (Baier and Winands 2012), where parameters specify the mean b and standard deviation c, so that b specifies where in the game play time is most concentrated and c specifies how concentrated play time is at the mean.

In our next experiment, we applied each time allocation policy to a simple player that selected the play with the best scoring average from uniformly-distributed Monte Carlo simulation with greedy score-maximizing play to a

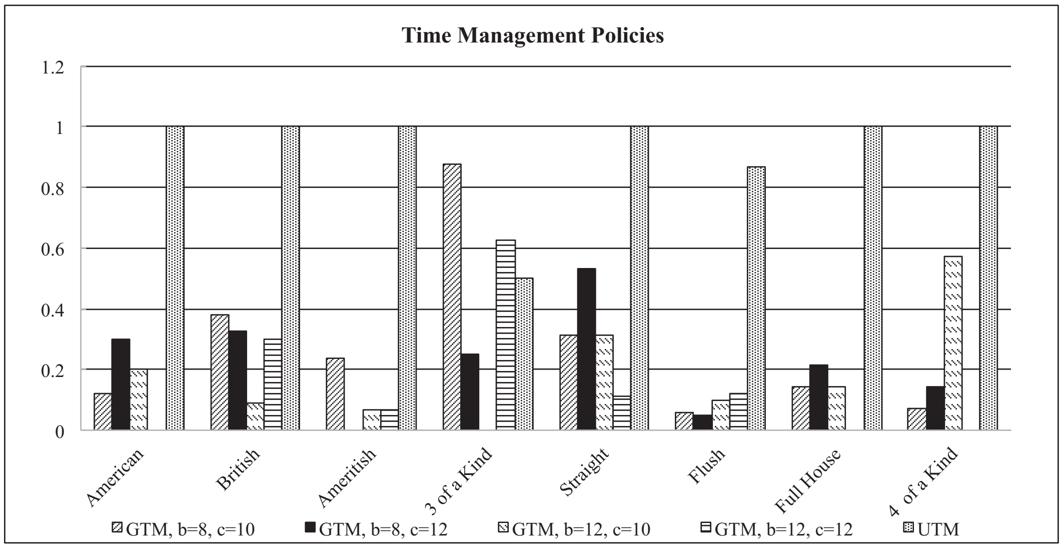

Figure 2: Comparison of performance under different time management policies. UTM was the best time management policy overall.

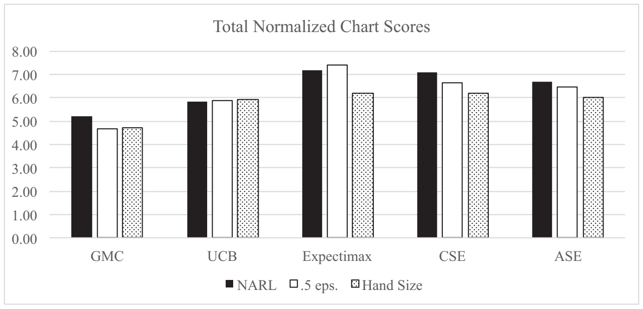

Figure 3: Comparison of performance of combinations of search algorithms and evaluators. Expectimax (depth 2) combined with the NARL evaluator ($\epsilon_0 = 0.5$) performed best.

depth limit of 5. GTM was tested with multiple players, with each player given different parameters for the curve's mean and standard deviation. Each player was tested for 100 games per point system. After all players had been tested for one system, they were assigned tournament points in the range from 0 to 1 as previously described. Results are shown in Figure 2.

UTM was the best time management policy overall. With the exception of the single-hand 3 of a kind and flush point systems, UTM always outperformed every other time management policy tested. For the single-hand flush system, UTM almost always outperformed, and for 3 of a kind, UTM was outperformed by GTM with b = 8, c = 10.

Search Algorithms

In order to make play decisions, the player uses a search algorithm to examine the branches of the game tree and determine the values of various moves. The player then makes play decisions based on the values of the states as determined by the static evaluation and the search algorithm. Several search algorithms were implemented, including Uniform Greedy Monte Carlo (GMC), UCB1 Greedy Monte Carlo (UCB), and depth-limited expectimax.

Uniform Greedy Monte Carlo *Greedy Monte Carlo* simulates play from the current state up to a given depth limit 5 ahead of each possible move, and then computes a static evaluation on the resulting state. Whereas the first action is taken a uniform number of times, following moves are chosen at random from among greedy plays that yield the maximum score for the next state. The move which returns the greatest average score at the depth limit or terminal state is then selected.

UCB1 Greedy Monte Carlo *UCB1* (Auer, Cesa-Bianchi, and Fischer 2002) looks to balance exploration of the game tree with exploitation of the best scoring opportunities. Implementation was much like Greedy Monte Carlo, except that the root actions were chosen with the UCB1 algorithm. The algorithm finds the average score resulting from a move in simulation, then adds the term $\sqrt{\frac{2ln(n)}{n_j}}$, where n is the total number of simulations and n_j is the number of simulations of a move j. In our version, the added term is multiplied by 20 to promote more frequent early exploration.

Expectimax *Expectimax* is the single-player specialization of expectiminimax (Russell and Norvig 2010, p. 178). Given our real-time constraints, expectimax could only be performed to a shallow depth limit of 2. Two variants were created in order to solve this problem: *chance-sampled expectimax* (CSE) takes a random sample of chance events and averages them, while *action-sampled expectimax* (ASE) uses the static evaluation to approximate values for each choice node and then performs full simulations of the best.

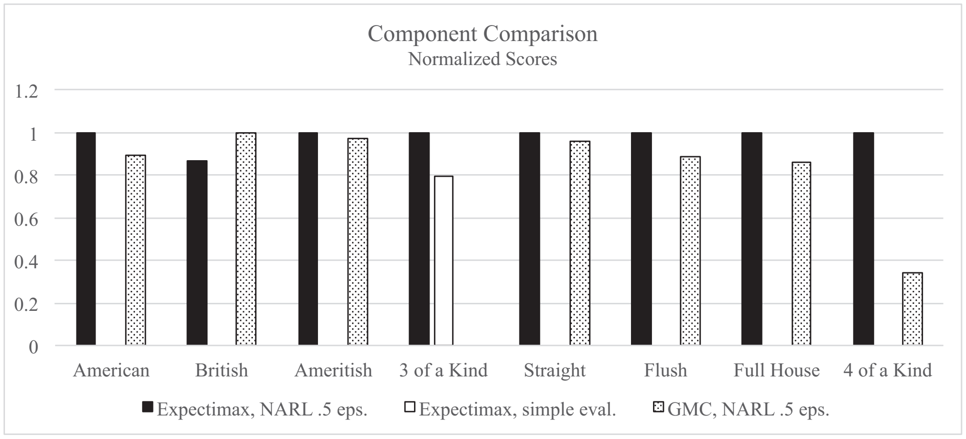

Figure 4: Comparison of individual component significance with various point systems. The learned static evaluation component was most significant.

Our action-sampled expectimax also made use of chance-sampling. These algorithms made no use of time management.

Experimental Comparison To compare the relative performance of 5 different search methods (Greedy Monte Carlo, UCB1, Expectimax (depth 2), Chance Sampled Expectimax (CSE; depth 2, 38 chance sample maximum), and Action Sampled Expectimax (ASE; depth 2, 40 chance samples maximum, 20 action samples maximum)) combined with the 3 different NARL learning systems we compared before, we ran a tournament with 8 scoring systems (Figure 3). The Ameritish system was generated with contest software using seed 21347. One hundred games were run per player and point system pair. UTM was used for all players that used time management. Total scores listed are the sum of the tournament scores for each of the 8 point systems.

In these experiments, expectimax (depth 2) combined with the NARL evaluator ($\epsilon_0 = 0.5$) performed best.

Significance of Each Design Component

After determining that expectimax search (depth 2) with NARL evaluation ($\epsilon_0 = 0.5$) was our best combination, a final test was performed to determine how significant each of the player's components was. Two additional players were constructed with components omitted. The first used depth 2 expectimax, but with a simple static current-score evaluation. The second used greedy Monte Carlo play combined with the NARL ($\epsilon_0 = 0.5$) evaluation. Time management was not tested as a factor, as the best search algorithm did not use time management. However, the greedy Monte Carlo player did utilize UTM for its necessary time management. For each point system and player pair, 100 games were played and scoring was performed by tournament rules. Results can be seen in Figure 4.

Reverting from Naïve Abstract Reinforcement Learning (NARL) to a simple scoring of the current grid caused the most significant decline in performance, winning a fraction of hands under the single-hand 3-of-a-kind point system, and performing worst at all other games. It is interesting to note that complete expectimax search to depth 2 only slightly outperformed greedy Monte Carlo to depth 5 for most point systems.

Coupling this with the observation that depth 2 expectimax only used about 2.8 seconds out of its allotted 30 seconds of play time per game, it is likely that our search time would be better utilized in future work through either a hybrid of the two approaches, e.g. greedy Monte Carlo estimation at expectimax depth 2, or through the application of a Monte Carlo Tree Search algorithm (Browne et al. 2012), an approach seeing much success across many similar games.

Conclusion

From our experimentation with reinforcement learning of a static state evaluator based on hand abstractions, various time management techniques for real-time play, and various search techniques, we found our static state evaluator work to be the most significant source of performance improvement.

This has two possible implications: Either (1) our easiest future performance gains could come from further improvement of the static evaluator, or (2) our greatest untapped potential is in the area of search algorithm design. Given that we fully utilize our static evaluator learning time and use only 9.4% of allotted play decision time, we conjecture that the greatest improvements to be realized will be in improved search algorithm design.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning* 47(2-3):235–256.

Baier, H., and Winands, M. 2012. Time management for monte-carlo tree search in go. In van den Herik, H., and Plaat, A., eds., *Advances in Computer Games*, volume 7168 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 39–51.

BoardGameGeek.com. 2015. *Poker Squares*. http://www.boardgamegeek.com/boardgame/41215/poker-squares.

Browne, C.; Powley, E.; Whitehouse, D.; Lucas, S.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of Monte Carlo tree search methods. *IEEE Trans. on Computational Intell. and AI in Games* 4(1). also available at http://www.cameronius.com/cv/mcts-survey-master.pdf.

Morehead, A. H., and Mott-Smith, G. 1949. *The Complete Book of Solitaire & Patience Games*. Grosset & Dunlap, 1st edition.

Neller, T. 2013a. *Poker Squares*. http://cs.gettysburg.edu/~tneller/games/pokersquares.

Neller, T. 2013b. *Sequential Placement Optimization Games*. http://www.boardgamegeek.com/geeklist/152237/sequential-placement-optimization-games.

Parlett, D. 2008. *The Penguin Book of Card Games*. Penguin Books, updated edition.

Russell, S., and Norvig, P. 2010. *Artificial Intelligence: a modern approach*. Upper Saddle River, NJ, USA: Prentice Hall, 3 edition.