Monte Carlo Approaches to Parameterized Poker Squares

Todd W. Neller[1](✉), Zuozhi Yang[1], Colin M. Messinger[1], Calin Anton[2], Karo Castro-Wunsch[2], William Maga[2], Steven Bogaerts[3], Robert Arrington[3], and Clay Langley[3]

[1] Gettysburg College, Gettysburg, PA, USA
tneller@gettysburg.edu
[2] MacEwan University, Edmonton, AB, Canada
antonc@macewan.ca
[3] DePauw University, Greencastle, IN, USA
stevenbogaerts@depauw.edu

Abstract. Parameterized Poker Squares (PPS) is a generalization of Poker Squares where players must adapt to a point system supplied at play time and thus dynamically compute highly-varied strategies. Herein, we detail the top three performing AI players in a PPS research competition, all three of which make various use of Monte Carlo techniques.

1 Introduction

The inaugural EAAI NSG Challenge[1] was to create AI to play a parameterized form of the game Poker Squares. We here describe the game of Poker Squares, our parameterization of the game, results of the competition, details of the winners, and possible future directions for improvement.

2 Poker Squares

Poker Squares[2] (a.k.a. Poker Solitaire, Poker Square, Poker Patience) is a folk sequential placement optimization game[3] appearing in print as early as 1949, but likely having much earlier origins. Using a shuffled 52-card French deck, the rules of [7, 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:

[1] Whereas DARPA has its "grand challenges", ours are not so grand.
[2] http://www.boardgamegeek.com/boardgame/41215/poker-squares, http://cs.gettysburg.edu/~tneller/games/pokersquares.
[3] http://www.boardgamegeek.com/geeklist/152237/sequential-placement-optimization-games.

© Springer International Publishing AG 2016
A. Plaat et al. (Eds.): CG 2016, LNCS 10068, pp. 22–33, 2016.
DOI: 10.1007/978-3-319-50935-8_3

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).

Note that the single remaining Poker hand classification of "high card", which does not fit any of the above classifications, scores no points.

3 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." [9, 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 very 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 as ordered in the table above, the player then plays Poker Squares according to the given point system.

The goal is to design Poker Squares AI with high expected score performance across the distribution of possible score parameters.

4 Point Systems

Contest point systems consisted of the following types.

– Ameritish - a randomized hybrid of American and English (a.k.a. British) point systems; includes American and English systems (given above)
– Random - points for each hand category are chosen randomly in the range $[-128, 127]$
– Hypercorner - points for each hand category are chosen with equal probability from $\{-1, 1\}$
– Single Hand - only one hand category scores 1 point; all other categories score no points

Hand categories are decided according to the rules of Poker, with higher ranking hand categories taking precedence. Note that the high card hand category may be awarded points in non-Ameritish systems.

4.1 Contest Structure and Results

For each point system tested in contest evaluation, each Java player program was given the point system and 5 min to perform preprocessing before beginning game play. For each game, each player was given 30 s of total time for play decision-making. A player taking more than 30 s of total time for play decision-making or making an illegal play scored 10 times the minimum hand point score for the game.

For each point system tested, each player's scores were summed to a total score and this total was normalized to a floating point number ranging from 0 (lowest score of all players) to 1 (highest score of all players). Players were ranked according to the sum of their normalized scores across all point system tests. All testing was performed on a Dell Precision M4800 running Windows 7 (64-bit) with and Intel Core i7-4940MX CPU @ 3.1 GHz, 32 GB RAM, and running Java version 1.8.0_51. Results of the contest can be seen in Fig. 1.

Players	Mean Scores by Point System											
	American	Ameritish	British	Hypercorner	Random	High Card	One Pair	Two Pair	3 of a Kind	Straight	Flush	Full House
BeeMo	125.27	105.54	54.50	1.10	437.77	9.37	9.12	4.46	3.20	2.97	3.43	1.82
DevneilPlayer	14.36	15.27	7.51	-9.52	-86.92	5.22	4.10	0.45	0.21	0.04	0.05	0.03
Gettysburg	123.94	110.28	53.38	1.24	429.89	9.37	9.17	4.47	3.02	2.71	3.46	1.93
SRulerPlayer	51.83	55.39	30.29	-5.10	242.85	9.34	8.84	4.04	2.10	1.58	1.98	0.61
JoTriz	116.75	109.03	53.59	-0.78	351.07	9.31	9.15	4.59	3.03	2.59	3.36	1.67
Tiger	116.12	111.26	53.92	-2.20	411.78	9.35	9.16	4.52	2.89	2.94	3.41	1.82
MonteCarloTreePlayer	15.47	15.31	7.61	-9.30	-86.83	4.80	4.53	0.45	0.20	0.05	0.02	0.00
RandomPlayer	14.25	15.67	7.71	-9.66	-106.80	5.20	4.31	0.42	0.23	0.01	0.01	0.01

	American	Ameritish	British	Hypercorner	Random	High Card	One Pair	Two Pair	3 of a Kind	Straight	Flush	Full House
Max	125.27	111.26	54.50	1.24	437.77	9.37	9.17	4.59	3.20	2.97	3.46	1.93
Min	14.25	15.27	7.51	-9.66	-106.80	4.80	4.10	0.42	0.20	0.01	0.01	0.00

Normalized Scores	American	Ameritish	British	Hypercorner	Random	High Card	One Pair	Two Pair	3 of a Kind	Straight	Flush	Full House	Total
BeeMo	1.00	0.94	1.00	0.99	1.00	1.00	0.99	0.97	1.00	1.00	0.99	0.94	**11.821**
DevneilPlayer	0.00	0.00	0.00	0.01	0.04	0.09	0.00	0.01	0.00	0.01	0.01	0.02	**0.190**
Gettysburg	0.99	0.99	0.98	1.00	0.99	1.00	1.00	0.97	0.94	0.91	1.00	1.00	**11.763**
SRulerPlayer	0.34	0.42	0.48	0.42	0.64	0.99	0.93	0.87	0.63	0.53	0.57	0.32	**7.149**
JoTriz	0.92	0.98	0.98	0.81	0.84	0.99	1.00	1.00	0.94	0.87	0.97	0.87	**11.170**
Tiger	0.92	1.00	0.99	0.68	0.95	1.00	1.00	0.98	0.90	0.99	0.99	0.94	**11.334**
MonteCarloTreePlayer	0.01	0.00	0.00	0.03	0.04	0.00	0.08	0.01	0.00	0.01	0.00	0.00	**0.192**
RandomPlayer	0.00	0.00	0.00	0.00	0.00	0.09	0.04	0.00	0.01	0.00	0.00	0.01	**0.153**

Fig. 1. Results of Contest Evaluation

Non-fixed point systems were generated with contest random seed 34412016. The twelve point systems used for contest evaluation included American, Ameritish, British,

Hypercorner, Random, and the following seven Single-Hand systems: High Card, One Pair, Two Pairs, Three of a Kind, Straight, Flush, and Full House.

Detailed performance information is available online[4]. Final contest standings were as follows:

1. Score: 11.821; Player: BeeMo; Students: Karo Castro-Wunsch, William Maga; Faculty mentor: Calin Anton; School: MacEwan University
2. Score: 11.763; Player: GettysburgPlayer; Students: Colin Messinger, Zuozhi Yang; Faculty mentor: Todd Neller; School: Gettysburg College
3. Score: 11.334; Player: Tiger; Students: Robert Arrington, Clay Langley; Faculty mentor: Steven Bogaerts; School: DePauw University
4. Score: 11.170; Player: JoTriz; Student: Kevin Trizna; Faculty mentor: David Mutchler; School: Rose-Hulman Institute of Technology
5. Score: 7.149; Player: SRulerPlayer; Student: Zachary McNulty; Faculty mentor: Timothy Highley; School: La Salle University
6. Score: 0.192; Player: MonteCarloTreePlayer; Student: Isaac Sanders; Faculty mentor: Michael Wollowski; School: Rose-Hulman Institute of Technology
7. Score: 0.190; Player: DevneilPlayer; Student: Adam Devigili; Faculty mentor: Brian O'Neill; School: Western New England University

As a benchmark, a random player was evaluated alongside contestants, scoring 0.153 tournament points. We first note that a cluster of 4 players scored close to the tournament maximum possible score of 12. The two bottom entries scored only slightly better than random play.

In the following sections, we will provide details of the top three performing players.

5 BeeMo

BeeMo implements a parallel flat Monte Carlo search guided by a heuristic which uses hand patterns utilities. These utilities are learned through an iterative improvement algorithm involving Monte Carlo simulations and optimized greedy search. BeeMo's development process was focused on three domains: game state representation, search, and learning. For each of these domains, we investigated several approaches. In the following subsections, we present the best combination of approaches according to empirical evaluations. For a more detailed description of all designs, see [4].

5.1 Game State Representation

We used a simple array representation for the tableau of face up cards and a bit packed representation for the deck of face down cards. We implemented a hand encoding scheme based on hand patterns, which are a representation of hands which retains only the relevant hand information:

- if the hand contains a flush - 1 bit;
- if the hand contains a straight - 1 bit;

[4] http://cs.gettysburg.edu/~tneller/games/pokersquares/eaai/results/.

- number of cards in the hand without a pair - 3 bits;
- number of pairs in the hand - 2 bits;
- if the hand contains three of a kind - 1 bit;
- if the hand contains four of a kind -1 bit;
- if the hand is a row - 1 bit.

The hand pattern encoding was extended with contextual information about the number of cards of the primary rank and secondary rank remaining in the deck. These are the ranks with the largest and second largest number of repetitions in the hand, respectively. We also added information about the number of cards in the remaining deck that can make the hand a flush. Instead of actual number of cards we used a coarse approximation with three values: not enough cards, exactly enough cards, and more than enough cards remaining to complete a certain hand. The approximation is represented on 2 bits, and so the contextual information adds 6 extra bits to the hand pattern, for a total of 16 bits for hand pattern and contextual information. The extra information increases the number of empirically observed unique patterns by a factor of 10. Our experiments indicate that this added complexity is overcome by the significant improvement in training accuracy.

5.2 Search

We implemented several game tree search algorithm classes: rule based, expectimax, greedy, and Monte Carlo. We compared their performance using a crafted heuristic for the American scoring system. Our empirical analysis indicated that the best search algorithms belong to the Monte Carlo and Greedy classes. As the final agent uses a combination of flat Monte Carlo and optimized greedy, we will present our implementation of these algorithms.

Optimized Greedy implements a greedy strategy based on hand pattern utilities. Every new card placed on the tableau of face up cards, influences only two hands: the column and the row in which the card is placed. Thus, the value of placing the card in any position can be estimated by the sum of the changes of the values for the column hand and for the row hand. The change in every hand is the difference of the hand pattern utility after and before placing the card. The optimized greedy algorithm places the card in the position that results in the largest value. Computing these values is very fast as it needs four look ups in a hash table of hand pattern utilities, two subtractions and an addition. It is exactly this simplicity that makes the algorithm very fast.

The algorithm plays ten of thousand of games per second, has impressive performances (consistently scores over 115 points on the American scoring), and it is essentially the same for any scoring system - the only changes are in the hand pattern utilities' hash table entries.

Flat Monte Carlo is a variation of the imperfect information Monte Carlo [5]. At a given node, the algorithm evaluates each child by averaging the scores of a large number of simulated games from that child, and then selects a move that results in the child with the largest value. Any search algorithm can be used to guide the simulated games, but a fast one is preferable. For this reason we used the optimized greedy search for the simulated games.

The resulting algorithm consistently outperformed optimized greedy by a margin of 10 points on the American scoring. However, the time efficiency of the algorithm was significantly worse than that of optimized greedy. Parallelization improved the algorithm speed significantly. In its final implementation the algorithm creates several threads of game simulations which are run in parallel on all available cores. Despite being an order of magnitude slower than the greedy algorithm, the parallel flat Monte Carlo is fast and has the best score performances of all the algorithms we tried.

5.3 Learning

Because the scoring scheme is not known, learning the partial hand utilities is the most important part of the agent. We used Monte Carlo simulations for learning the hand pattern utilities. The learning algorithm uses rounds of 10,000 Monte Carlo simulations, 5,000 for training and 5,000 for evaluation.

All hand pattern utilities are initialized to zero. At the end of a training game, the value of each final hand is recorded for every hand pattern which resulted in the hand. For example, if a partial hand has only a 5♥ and at the end of the game results in a flush with a score of 20, then a value of 20 is recorded for the pattern which encodes a hand with only 5♥. The utility of a hand pattern is estimated as the average of all recorded values. The updated hand pattern utility set is used in the next simulated game.

Using flat Monte Carlo search for the training phase, the agent learned hand pattern utilities which for the American scoring resulted in performance comparable to those obtained using the crafted hand pattern utilities, and reduced running time.

While simple and fast, flat Monte Carlo search suffers from lack of specialization. When used for learning hand pattern utilities this drawback negatively affects the accuracy of the estimations. For example, low frequency patterns with high utilities are rarely updated and thus the learned values may be unreliable. To check if our learning algorithm has such a pitfall we implemented a UCT evaluation scheme inspired by the UCB1 variation of the Upper Confidence Bound [2]. We used a small exploration parameter which was optimized empirically. UCB1 slightly increased the number of discovered patterns, but its influence on agent's performance was positive only for the American and British scoring systems. For the final agent we decided to use both evaluation schemes by alternating them with different frequencies.

In the evaluation phase, 5,000 simulated games are played using the set of hand pattern utilities learned in the training phase. The average of the games' scores is used to evaluate the overall utility of a set of patterns. The set with the highest overall utility is used by the final agent. The agent consistently completes 180 rounds of learning during the allowed 300 s. However, most of the improvements are done in the first 10 rounds, after which the performance evolution is almost flat.

As indicated in the contest results, the final agent played strongly under all scoring systems. Given that (1) players employed various heuristics and differing uses of Monte Carlo techniques, and (2) players achieved similar peak performance, we conjecture that these top players closely approximate optimal play.

6 GettysburgPlayer

The GettysburgPlayer uses a static evaluation, which abstracts game states and attempts to assess their values given any scoring system, in combination with expectimax search limited to depth 2.

6.1 Static Evaluation

The total state space is too large to evaluate in advance, so the state space is abstracted and on-policy Monte Carlo reinforcement learning is applied 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 continuously 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. However, purely random play makes certain types of hands highly improbable (e.g., royal flush, straight), so sufficient exploitation-heavy play time is necessary to learn the value of long-term attempts to achieve such hands.

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 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 (Fig. 2).

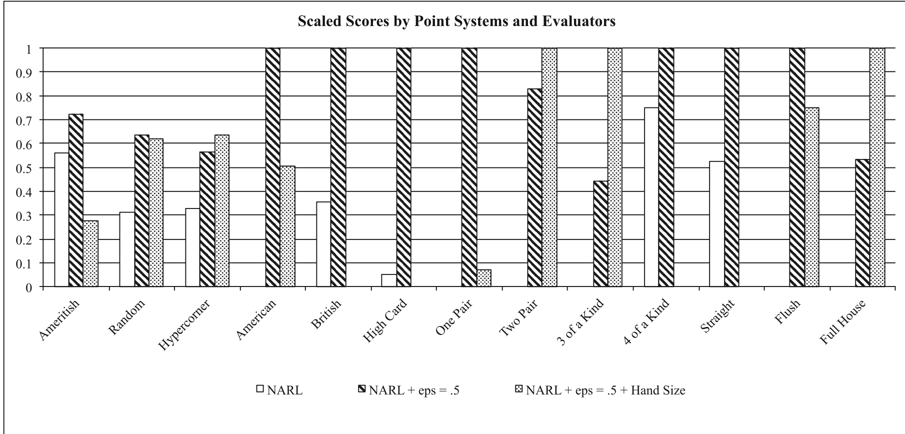

Fig. 2. Comparison of learning evaluation performance. NARL with $\epsilon_0 = 0.5$ performed best for most point systems.

6.2 Search Algorithm

Using the NARL static evaluation, we compared three search algorithms: (1) Flat Monte Carlo [3, Sect. 2.3] limited to depth 5, (2) Flat UCB1 limited to depth 5 and multiplying the exploration term $\sqrt{2ln(n)/n_j}$ by 20 to encourage greater exploration[5], and (3) expectimax with a depth limit of 2.

The three algorithms were paired with the three static evaluators and tested against each other using the contest software, 8 scoring systems, and the seed 21347. Each player ran 100 games per point system. The final comparison was based upon each player's total normalized score. A combination of depth 2 expectimax and the NARL evaluator ($\epsilon_0 = 0.5$) received the highest total score and was submitted for competition.

Significance testing of the various player components revealed that our static evaluation function was most significant to the player's performance [8]. Space limitations preclude the full set of alternative designs considered. However, these and relevant experimental data are available in [8] as well.

7 Tiger: A Heuristic-Based MCTS Player

This summary is based on a more detailed discussion available in [1]. The player uses Monte Carlo Tree Search (MCTS) [3] with added domain knowledge to select moves.

7.1 Design and Application of the State Heuristic

This player includes a state heuristic that can accommodate any scoring system. It can be framed as ten applications of a hand heuristic, corresponding to the ten rows and

[5] The factor of 20 was chosen through limited empirical performance tuning. It is not necessarily optimal for this problem.

columns in the game. Five-card hands are simply scored according to the current scoring system. One to four card hands, however, are evaluated with probability estimates.

In four-card hands, for each hand type, a *weight* in [0, 1] is calculated, representing an estimated likelihood of obtaining that hand type with the next card draw given the cards remaining in the deck. For example, suppose a 4-card hand contains a 6♥, 6♣, 6♠, and 7♠, while the deck contains only a 6♦, 7♥, and 8♥. Here, three-of-a-kind is given a weight of 1/3 because among the remaining cards only 8♥ would result in three-of-a-kind. Once these weights are calculated for every hand type, each weight is multiplied by the hand type value according to the current scoring system, and added together to form a weighted sum. Note that this ignores the fact that the ten hands in the grid are dependent on each other, both spatially and in "competition" for cards.

This approach gets much more computationally intensive for hands with fewer than four cards, and so in this case we instead use estimated *a-priori* probabilities of hand types as weights. These probabilities are then used to compute a weighted sum in the same way as in a four-card hand. Note, however, that by this measure hands with fewer cards will be inadvertently favored, because fewer cards will tend to mean more possible hand-types. To counter this, we apply weights α, β, and γ to one, two, and three-card hand heuristic values, respectively. For now, we fix these values at $\alpha = 0.2$, $\beta = 0.4$, and $\gamma = 0.6$, with tuning experiments described below.

With this heuristic, various selection strategies exist. *UCT* [6] is a standard measure balancing exploration and exploitation with no domain-specific heuristic. *Best Move* always chooses the single unexplored node with the highest heuristic value. *Prune + UCT* prunes nodes below a heuristic score threshold and then applies standard UCT.

In simulation, *Random* is the standard MCTS strategy of choosing random moves without a heuristic. *Prune + Random* chooses moves at random from a tree pruned via the heuristic. *Best Move* chooses the single move with the highest heuristic value.

7.2 Experiments and Results

Table 1 shows results for various experiments. We begin by considering the practical cost of calculating the heuristic itself, since time spent on heuristic calculations means fewer iterations of the core MCTS process. Row (1) reflects standard MCTS, with UCT selection and Random simulation. Rows (2) – (4) also use these standard strategies, but with the "+ Calc." notation indicating that the heuristic calculations are *performed* but not actually *applied*. Note a total cost of 13 points (comparing rows (1) and (4)) in the American scoring system when the heuristic is calculated in both selection and simulation, with most of the cost coming from simulation.

In simulation, ignoring for now the "Tuned" column, note that Prune + Random's score of 95 (row (5)) shows improvement over the 92 of standard MCTS (row (1)) and the 80 of the added heuristic calculation cost (row (3)). Best Move simulation (row (6)) improved more strongly to an untuned score of 112. It seems intuitive that Best Move simulation is more effective than Prune + Random, since Best Move plays a simulated game according to the best choices that the heuristic is capable of suggesting. In contrast, Prune + Random gives the heuristic less control, only determining a set of higher-scoring moves from which a random selection is made.

Table 1. Mean scores over 2,000 games for various selection and simulation strategies

Row	Selection	Simulation	Untuned	Tuned
(1)	UCT	Random	92	
(2)	UCT + Calc	Random	90	
(3)	UCT	Random + Calc	80	
(4)	UCT + Calc	Random + Calc	79	
(5)	UCT	Prune + Random	95	
(6)	UCT	Best Move	112	118
(7)	Best Move	Random	72	
(8)	Prune + UCT	Random	91	94
(9)	Prune + UCT	Best Move	113	123

Next consider the selection strategy results, again ignoring for now the "Tuned" column. Prune + UCT seems unhelpful when comparing rows (1) vs. (8), and (6) vs. (9) (untuned). The final set of experiments will consider this further. Best Move selection, in contrast, appears not merely unhelpful but *harmful*. With a score of 72 (row (7)), it scores even worse than row (2) in which the calculations are performed but not applied. This is not surprising, since such a drastic approach severely limits the number of nodes available for exploration. That is, while Best Move *simulation* is a useful limitation in contrast to Random simulation, the more principled exploration of *selection* with *UCT* should not be so severely restricted by a Best Move approach.

Finally, consider further tuning of α, β, and γ values for weighting one-, two-, and three-card hands, respectively. After experiments on many combinations of settings, it was found that $\alpha = 0.1, \beta = 0.3, \gamma = 0.85$ gave the best performance on the American scoring system, with other high-scoring settings converging on those values. Results for this setting are shown in the "Tuned" column of Table 1. This newly-tuned heuristic sheds new light on Prune + UCT selection, which seemed ineffective in the untuned results. Row (8) shows that Prune + UCT selection with tuned parameter settings attains a 94, compared to the 91 with the untuned settings, and the 92 (row (1)) of standard MCTS. Similarly, Best Move simulation now scores 118 (row (6)), showing further improvement over the untuned 112. These experiments demonstrate that both Prune + UCT selection and Best Move simulation can be improved and are worthwhile after tuning the heuristic, with a final top score of 123 when both strategies are applied.

8 Conclusion

The inaugural EAAI NSG Challenge was reported to be a very positive experience by both students and faculty. Informal evaluation indicates that more than half of entries perform well beyond human-level play, and most were densely clustered at the top of the distribution, lending confidence to a conjecture that optimal play is not far beyond the performance observed.

In the future, it would be interesting to perform more significance testing across implementations in order to demonstrate the relative value of different design components, e.g., the parallelization of BeeMo. Testing comparable elements of designs would guide a hybridization of approaches, e.g., testing a single search algorithm with each of our various static evaluation functions. We conjecture that an ensemble or hybrid approach would yield performance improvements.

References

1. Arrington, R., Langley, C., Bogaerts, S.: Using domain knowledge to improve monte-carlo tree search performance in parameterized poker squares. In: Proceedings of the 30th National Conference on Artificial Intelligence (AAAI 2016), pp. 4065–4070. AAAI Press, Menlo Park (2016)
2. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Mach. Learn. **47**(2–3), 235–256 (2002). http://dx.doi.org/10.1023/A:1013689704352
3. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. IEEE Trans. Comput. Intell. AI Games **4**(1), 1–49 (2012). http://www.cameronius.com/cv/mcts-survey-master.pdf
4. Castro-Wunsch, K., Maga, W., Anton, C.: Beemo, a Monte Carlo simulation agent for playing parameterized poker squares. In: Proceedings of the 30th National Conference on Artificial Intelligence (AAAI 2016), pp. 4071–4074. AAAI Press, Menlo Park (2016)
5. Furtak, T., Buro, M.: Recursive Monte Carlo search for imperfect information games. In: 2013 IEEE Conference on Computational Inteligence in Games (CIG), Niagara Falls, ON, Canada, 11–13 August 2013, pp. 1–8 (2013). http://dx.doi.org/10.1109/CIG.2013.6633646
6. Kocsis, L., Szepesvári, C., Willemson, J.: Improved monte-carlo search. Univ. Tartu, Estonia, Technical report 1 (2006)
7. Morehead, A.H., Mott-Smith, G.: The Complete Book of Solitaire & Patience Games, 1st edn. Grosset & Dunlap, New York (1949)
8. Neller, T.W., Messinger, C.M., Zuozhi, Y.: Learning and using hand abstraction values for parameterized poker squares. In: Proceedings of the 30th National Conference on Artificial Intelligence (AAAI 2016), pp. 4095–4100. AAAI Press, Menlo Park (2016)
9. Parlett, D.: The Penguin Book of Card Games. Penguin Books, updated edn. (2008)