Generating Heuristics for Novice Players

Fernando de Mesentier Silva, Aaron Isaksen, Julian Togelius, Andy Nealen
Tandon School of Engineering, New York University

***Abstract*—We consider the problem of generating compact sub-optimal game-playing heuristics that can be understood and easily executed by novices. In particular, we seek to find heuristics that can lead to good play while at the same time be expressed as fast and frugal trees or short decision lists. This has applications in automatically generating tutorials and instructions for playing games, but also in analyzing game design and measuring game depth. We use the classic game Blackjack as a testbed, and compare condition induction with the RIPPER algorithm, exhaustive-greedy search in statement space, genetic programming and axis-aligned search. We find that all of these methods can find compact well-playing heuristics under the given constraints, with axis-aligned search performing particularly well.**

I. Introduction

Much artificial intelligence and game theory research is focused on quickly finding optimal moves in games; however discovering optimal moves is often not practical for human players. Novices look for ways to understand basic concepts, players may have a limited amount of information in their working memory, experts may try to trick their opponents using gambits or other risky moves, and most games are just too complicated for humans (or machines) to evaluate in real-time to make optimal moves. In this paper, we focus on algorithmically generating simple introductory heuristics for novice players.

Nonetheless, novices are not the only types of players who use simplified models to make decisions. Instead of humans acting as purely rational agents making optimal decisions, the behavioral economic theory of *bounded rationality* claims that people make decisions based on a limited amount of available information and decision-making time [1], [2]. The process of making best guesses instead of optimally rational decisions is called *satisficing*. By using simple heuristics to make decisions, accuracy can actually improve over more complicated algorithms because simple guidelines are easier to execute without errors [3].

Teaching beginners a good simple strategy for a new game can be a challenge but essential for enjoyment of the game. Winning a match gives a sense of pleasure, but learning how to play and improving can lead to a sense of accomplishment [4]. A game in which it is difficult to learn basic strategies may be overwhelming for new players; a game in which high-performing strategies are easy to come up with may not be entertaining in the long run. Strategies that lead to moves that are effective, simple to execute, easy to remember, and allow the player to further improve are ideal. Examples of simple heuristics for well-known games include playing on the middle and corners before the sides in Tic-Tac-Toe, while in Chess a popular heuristic is learning the relative value of the pieces or simple opening moves to begin the game.

Many well-respected games allow the player to “climb a heuristic ladder” in which they learn deeper and more complicated strategies [5]. The length of the heuristic ladder can give a sense of a game’s depth. Strategies used by beginner players at Chess may be very different from those used by professionals. Players look for others of the same skill level to play against, and as they gather more experience their gameplay improves leading to more sophisticated and effective moves. Strategies that once looked overwhelming and confusing can become accessible. Games such as Chess and Go have such a high-dimensional game state space that they permit strategies only accessible to players of significantly higher skill levels, measurable in Go by the Kyu and Dan rankings and in Chess by Elo and Titles. On the other hand, easier games like Tic-Tac-Toe are more accessible for new players for having relatively simple and effective moves as a result of a low-dimensional space of strategies. Players of Tic-Tac-Toe don’t need years of training to improve their strategies to reach an optimal strategy; therefore they usually stop playing it after learning or discovering the optimal strategy as there is no longer any reason to learn and improve.

A Fast and Frugal Tree (FFT) is a particularly good form of human-usable heuristic and is commonly used in bounded rationality theory. FFTs are easy to process and in practice have very good performance over more complicated algorithms [6]. A FFT is a type of binary decision tree where at each decision node, one path leads to a terminal action and the other path either leads to a fast and frugal sub-tree or a default action, as shown Figure 1. These trees can also be implemented as series of if/elseif/else statements or as a decision list [7], [8]. In this paper, we generate FFTs that can be used by beginners for effectively playing the game of Blackjack.

In this work we describe some techniques to generate easy to understand and effective fast and frugal heuristics. We evaluate and compare the results produced by the various methods. Our test case game is Blackjack, as a powerful heuristic called Basic Strategy is already known for the game. This allows us to verify that our results agree with existing theory on Blackjack [9]. Additionally, Blackjack is a two player game where the second player (dealer) must always play by a pre-determined algorithm only after the first player has completed their actions – this makes it easier to analyze than games where both players take turns and are more free in their action selection.

Expert agents to efficiently play games have been explored for many games, including Chess [10], Othello [11], Checkers [12] and Go [13]. These agents are meant to win as often as possible given an efficient amount of computation, and in many

This work was done with the support of CAPES, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil.

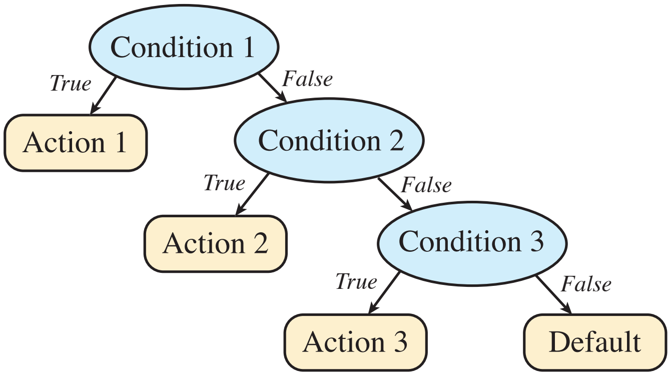


Fig. 1. Fast and Frugal Trees are a type of binary decision tree used in bounded rationality theory. They are effective and easy to learn heuristic for humans. In this paper, we generate fast and frugal trees – also known as decision lists – for beginning strategy for Blackjack.

cases they are able to compete on the level of world champion human players. The game playing heuristics we introduce in this paper are of much lower skill level as we expect a trade-off between performance and simplicity. The closer these simple heuristics get to optimal play in terms of expected value, the less need a player has to improve, and thus could possibly be used as a metric to estimate the potential depth of a game.

Evolutionary algorithms are especially effective at developing strategies for playing games at a high-performing level. Genetic Programming was used to evolve players for traditional adversarial board and card games such as Chess endgame [14], Lose Checkers [15], Backgammon [16] and Poker [17]. Adversarial video games such as Core War [18] and Robocode [19] incorporate evolved agents. Other work on evolving strategies include: Evolution of neurons for a neuro-network using reinforcement learning to generate controllers capable of playing levels on a Super Mario Bros clone [20], using genetic programming to create players for a solo variant of the game Pong [21], evolving agents to play the game Pac-Man [22] and evolving controllers with general driving skills [23]. Blackjack strategies have been evolved using strategy tables [24], genetic programming trees [25], and neural networks [26], [27] but these were focused on improving expert play, not focused on simple guidelines for novices. Blackjack AIs have been used to model and predict human gambling behaviors [28], in particular the ways that humans process and recall information [29].

Similarly to Hyper-heuristics [30] we search in heuristic space, however we only search for novice-friendly heuristics for a single problem, instead of switching between heuristics.

In the following sections, we first describe the rules of Blackjack and the Fast and Frugal Tree format we use for generating novice heuristics. The next four sections each describe a particular method for generating compact heuristics, as well as the results from applying these algorithms and heuristics to Blackjack. Finally we discuss the takeaways of this comparative study, as well as the prospects for generalizing the method to other games.

II. Blackjack

For evaluating different methods for generating fast and frugal heuristics for novice players, we selected Blackjack as a study case. Blackjack is a well-known gambling game played all over the world. It uses one or more standard 52 card decks, but 8 decks is typical in modern casinos. In this paper we assume an infinite number of decks to avoid issues with counting cards or distributional effects caused by drawing without replacement.

Blackjack is played against a dealer. The player tries to build a hand of cards that beats the dealer's hand, without going over 21 points. The game starts with a player betting and then two cards are dealt to the player while the dealer gets one card face down and one face up. Play proceeds with the player making all their moves followed by the dealer.

The player has 4 actions to choose from: *hit*, the player receives the next card from the deck; *stand*, the player stops and plays proceeds to the dealer; *double down*, the player doubles their bet, receives exactly one more card and then plays proceeds to the dealer; and *split*, the player splits their two cards into two new hands and the player matches the initial bet for the new second hand. A player can only split when they have 2 cards and they are of the same value. The player loses immediately if they goes over 21 points, called a *bust*. After hitting or splitting, the player can continue to make more actions until they bust or stand.

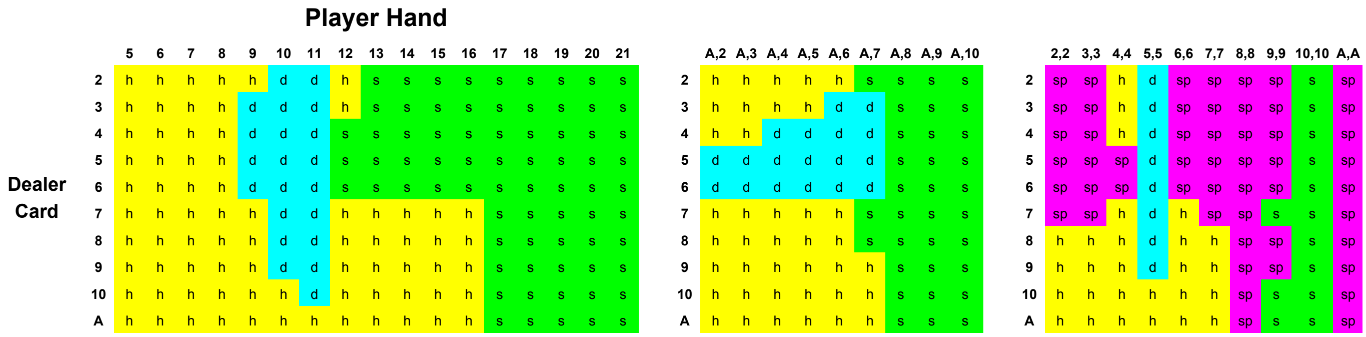
Cards in Blackjack are worth their face value, independent of their suit, with the exception of the Jack, Queen and King which are all worth 10 points and the Ace which is worth either 1 or 11. The Ace takes on the value of whichever makes the hand worth the most amount of points without busting. If the hand contains an Ace that can still change value from 11 to 1 it is called *Soft*.

After the player made all their moves for a hand, play proceeds to the dealer. The dealer plays his moves following an algorithm: they *hit* while the total amount of points in their hand is below 17. When they reaches 17 or more they *stand*. If the dealer's score exceeds 21 the dealer busts and the player wins the hand (if they have not already busted).

There is one special case: when a hand is composed by an Ace and one other card that is worth 10 points it is considered a Blackjack. A Blackjack beats any other hand of 21 or less, except another Blackjack, which results in a tie.

In Blackjack the dealer, who plays for the casino, has the advantage over the player. That mainly comes from the fact he plays last and so the player can lose the game by *busting* without the dealer ever having to make a move. Although the game has an element of randomness in the card drawing, Blackjack has some skill [31]. Strategies for playing the game to reduce the dealer's advantage go from what is called Basic Strategy [9] or Optimum Strategy [32], which closely relates to the work we do in this paper, to more complex such as card counting, which we do not address as it is not for novice play.

The Basic Strategy shown on Figure 2 determines the best move for the player over the hand of the player and the card the dealer is showing. Special cases occur when the player holds an Ace or a hand that can be split.



Player Hand

Dealer Card	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
2	h	h	h	h	h	d	d	h	s	s	s	s	s	s	s	s	s
3	h	h	h	h	d	d	d	h	s	s	s	s	s	s	s	s	s
4	h	h	h	h	d	d	d	s	s	s	s	s	s	s	s	s	s
5	h	h	h	h	d	d	d	s	s	s	s	s	s	s	s	s	s
6	h	h	h	h	d	d	d	s	s	s	s	s	s	s	s	s	s
7	h	h	h	h	h	d	d	h	h	h	h	h	s	s	s	s	s
8	h	h	h	h	h	d	d	h	h	h	h	h	s	s	s	s	s
9	h	h	h	h	h	d	d	h	h	h	h	h	s	s	s	s	s
10	h	h	h	h	h	h	d	h	h	h	h	h	s	s	s	s	s
A	h	h	h	h	h	h	h	h	h	h	h	h	s	s	s	s	s

	A,2	A,3	A,4	A,5	A,6	A,7	A,8	A,9	A,10
2	h	h	h	h	h	s	s	s	s
3	h	h	h	h	d	d	s	s	s
4	h	h	d	d	d	d	s	s	s
5	d	d	d	d	d	d	s	s	s
6	d	d	d	d	d	d	s	s	s
7	h	h	h	h	h	s	s	s	s
8	h	h	h	h	h	s	s	s	s
9	h	h	h	h	h	h	s	s	s
10	h	h	h	h	h	h	s	s	s
A	h	h	h	h	h	h	s	s	s

	2,2	3,3	4,4	5,5	6,6	7,7	8,8	9,9	10,10	A,A
2	sp	sp	h	d	sp	sp	sp	sp	s	sp
3	sp	sp	h	d	sp	sp	sp	sp	s	sp
4	sp	sp	h	d	sp	sp	sp	sp	s	sp
5	sp	sp	sp	d	sp	sp	sp	sp	s	sp
6	sp	sp	sp	d	sp	sp	sp	sp	s	sp
7	sp	sp	h	d	h	sp	sp	s	s	sp
8	h	h	h	d	h	h	sp	sp	s	sp
9	h	h	h	d	h	h	sp	sp	s	sp
10	h	h	h	h	h	h	sp	s	s	sp
A	h	h	h	h	h	h	sp	s	s	sp

Fig. 2. The Basic Strategy table. *s=stand*, *h=hit*, *d=double down* and *sp=split*. By our definition, the table has a fitness of 0 (expected loss of -0.0051875).

Despite being simple to read, the Basic Strategy has several levels of granularity and so it can be hard to remember all possible scenarios. Additionally, referring to the card while playing is allowed but error prone, especially for beginner level players. In the next sections we present several methods for generating simple novice level heuristics for Blackjack.

III. Heuristic representation

The simple human-playable heuristics we generate can be represented as fast and frugal trees, decision lists, or as a series of if/elseif/else statements. For compactness, we use the if/elseif/else format for this paper but they are all equivalent. The heuristics are composed of any number of condition/action pair and a default action. Each condition/action pair is represented as if/elseif-statements. Conditions are formed of one or more clauses that must all be true for a condition to be true. The default action is taken if all the conditions are not satisfied, and is represented by an else-statement.

Each clause is permitted to analyze P_{pts}, the total number of points currently held by the player, and D_{pts}, the number of points on the dealer's visible card. P_{pts} can be compared to π_{lower} and π_{upper}, parameterized lower and upper bounds for P_{pts}. δ_{lower} and δ_{upper} are parameters for lower and upper bounds for the clauses matching D_{pts}.

Two conditional boolean statements can also be checked, $canSplit$ and $isSoft$. If a player's hand has two cards and they are both of the same value, $canSplit$ is $True$, $False$ otherwise. If there is still an Ace in the player's hand that can change value, $isSoft$ is $True$, on any other case it is $False$.

With such, our heuristics are formed as a set of statements with the following structure:

```
if CONDITION 1 then ACTION 1
else if CONDITION 2 then ACTION 2
else if ... then ...
else DEFAULT ACTION
```

where each condition is formed by conjoining together one or more clauses selected from $\pi_{lower} \leq P_{pts}$, $P_{pts} \leq \pi_{upper}$, $\delta_{lower} \leq D_{pts}$, $D_{pts} \leq \delta_{upper}$, $canSplit$, $not\ canSplit$, $isSoft$ and $not\ isSoft$.

We define the complexity of a heuristic by summing up the total number of clauses plus the number of actions plus 1 for the default action. This coarse definition does not take into account that smaller numbers might be easier to memorize than longer ones, or some orderings of conditions might be easier to recall. However, our definition allows us a simple way of comparing more complex heuristics with simpler ones.

We define fitness to be the expected value of the heuristic minus the expected value of Basic Strategy.

IV. Methods for Fast Simulation

In order to evaluate the quality of a heuristic, we need to test it against a large number of possible hands. We are thus incentivized to make our simulation run as fast as possible. We do this by (1) precomputing as many results as possible and (2) parallelizing the calculation.

Dealer will act only after having complete information of what the Player's actions have been. Therefore, given (1) the final sum of the Player's cards, (2) assuming an infinite deck (i.e. choosing with replacement) where the distribution of cards is the same for every draw, and (3) a fixed algorithm that the Dealer must follow on their turn, we can precompute all possible plays for the Dealer once the Player has finished their turn. This precomputation allows us to store the expected value of a Player's final score given the Dealer's visible card.

We calculate the table of expected values as follows. Given every possible hand of cards for the Dealer and final score for the Player, there are a total of 79,489 situations that might occur. For each possible hand, we know if the game is a win (+1), tie (0), or loss (-1) given a Player's final score. For each final score for the Player and face-up Dealer card, we calculate the expected value by summing the number of wins minus losses, dividing by the total number of possible hands given that first card of the Dealer. We store these expected values in a table. Instead of playing the Dealer hands, we just look up the expected value from the table and use the precomputed result of stopping with that player score against the dealer's visible card. Double downs need to multiply the expected value by 2, and Blackjacks for the Player count as 1.5 wins instead of 1. Splits require special handling, because the outcome of the Player's two hands are not independent. Given the Player's two final scores from the split, we calculate the expected value of the hands together and store this in a separate table.

Unfortunately, we can't precompute the Player's hands because the Player is able to stop at any time which gives rise to too many possible hands to iterate through (and some of them are extremely unlikely to occur, such as a hand with ten 2's and one ace). However, we can be sure to deal out all possible starting 2 cards for the Player and 1 starting card for the Dealer to ensure the simulations cover a wide range of the most likely outcomes. For each of these 13*13*13 = 2,197 starting conditions, we simulate between 100 and 500 games, depending on the speed and accuracy required.

Because every game is independent, we can parallelize the simulation across multiple cores. We use the python multiprocessing library to split up the calculation across cores on a single computer. In addition, we use Cython to compile the simulation steps into faster C++.

Finally to avoid sampling errors where running equivalent strategies would give different results depending on which cards were drawn, we use the same seed for every simulation. This ensures that equivalent strategies will get exactly the same fitness every time we run the simulation.

V. Inducting heuristics from the Basic Strategy

Following our expressions format we can generate decision lists [7] as our heuristics, as long as we have a database to extract our expressions from. For every possible initial configuration of the game, between the player's and dealer's hands, we simulated 200,000 games of Blackjack for each of the possible moves for the player's hand. Starting from the games with the higher starting hand value and working in descending order to the games with the lowest value, we tried to generate a table analogous to the Basic Strategy. From the average score of the playouts, the move with highest score was selected. By building the table in descending order of points, we could reference previous results to decide the next action to take after a move draws a new card. The database has 550 entries. We compared the results to the moves that would have been picked by the Basic Strategy. Both pick the same move for 94% of the entries in the data. The case in which they differ were mostly when picking *split* as a move with the Basic Strategy. With further analysis, we could tell that increasing the number of simulations would make the diverging data points converge to the Basic Strategy selections. For this reason, we decided to generate a database in the same format, but using the move selection according to the Basic Strategy.

To extract the heuristics from the database we use the Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [33] algorithm. The algorithm uses a grow and prune approach, followed by a revision stage. The database is split into two, $\frac{2}{3}$ is used in the grow step and the other $\frac{1}{3}$ in the prune step. For each class in the data, from the least prevalent to the most, conditions to classify that class are grown from the database by adding clauses until maximum information gain is reached. Pruning is then done on the condition to maximize a target function. The revision stage then analyzes each condition, in the order they were learned, and generate two new candidates for it. One, the replacement, is obtained by growing and pruning a new condition, where pruning looks to minimize the error on the set where the condition was replaced by this new candidate. The other, the revision, is generated by greedily adding more clauses to the condition. Finally a decision is made based on a heuristic to decide which to keep, the original condition, the replacement or the revision. To the conditions a resulting action is added. For Blackjack, the possible moves represent the classes. The most prevalent class, in this case *hit*, is used as the default move.

```
if $canSplit$ and $11 \leq P_{pts} \leq 18$ and $D_{pts} \leq 6$ then
    SPLIT
else if $canSplit$ and $isSoft$ then SPLIT
else if $canSplit$ and $P_{pts} \leq 6$ and $D_{pts} \leq 7$ then
    SPLIT
else if $canSplit$ and $P_{pts} \leq 8$ and $5 \leq D_{pts} \leq 6$ then
    SPLIT
else if $canSplit$ and $13 \leq P_{pts} \leq 18$ then SPLIT
else if $10 \leq P_{pts} \leq 11$ and $D_{pts} \leq 9$ then
    DOUBLEDOWN
else if $isSoft$ and $P_{pts} \leq 18$ and $5 \leq D_{pts} \leq 6$ then
    DOUBLEDOWN
else if $9 \leq P_{pts} \leq 11$ and $3 \leq D_{pts} \leq 6$ then
    DOUBLEDOWN
else if $isSoft$ and $17 \leq P_{pts} \leq 18$ and $3 \leq D_{pts} \leq 4$
then
    DOUBLEDOWN
else if $17 \leq P_{pts}$ then STAND
else if $10 \leq P_{pts}$ and $D_{pts} \leq 6$ then STAND
else HIT
```

Fig. 3. The heuristic generated using the RIPPER algorithm. The set correctly classified 92% of the entries in the database used. It has a fitness of -0.0134.

The results are shown in figure 3. The algorithm came up with a set of 11 statements, plus the default move. To simplify readability of the statements, we do a post-processing of the results: reduce the set of clauses in the conjunctions of a condition, if there are clauses whose coverage is already part of another clause in the same condition; reorder the clauses for readability, ordered from lower to upper bound coverage. Since one of our goals was to reduce the granularity of heuristic complexity in respect to the basic strategy, we believe that the algorithm successfully achieves that up to a point. The set generated misclassified 8% of the entries, meaning that it chooses a different move that contained in the database. By looking at the proximity in fitness, the misclassified scenarios have a very low impact on the outcome.

In terms of generating novice level heuristics, the one inducted by RIPPER is unwieldy. The conditions containing *canSplit* and *isSoft* together with the number of statements appears to be convoluted for a beginner. To address such, we decided to remove *isSoft* from the possible clauses and run the algorithm again, but we were still left with 8 rules and too much complexity. So we ran the algorithm once more removing both *isSoft* and *canSplit* from the possible clauses. Figure 4 shows the new heuristic, which is a lot closer to our goal, having cleaner conditions and almost half the total statements.

As the algorithm goes, it looks to generate conditions to classify members of each class, from the least prevalent to the most. Since the statements that make up the heuristic the algorithm generates start from least used moves, the later statements have much more impact on improving the score for the player. Since the heuristic generated by RIPPER has

```
if $P_{pts} \leq 4$ and $D_{pts} \leq 7$ then SPLIT
else if $10 \leq P_{pts} \leq 11$ and $D_{pts} \leq 10$ then DOUBLEDOWN
else if $9 \leq P_{pts} \leq 10$ and $4 \leq D_{pts} \leq 6$ then
    DOUBLEDOWN
else if $13 \leq P_{pts}$ and $D_{pts} \leq 6$ then STAND
else if $17 \leq P_{pts}$ then STAND
else if $10 \leq P_{pts}$ and $4 \leq D_{pts} \leq 6$ then STAND
else HIT
```

Fig. 4. Using the RIPPER algorithm the heuristic generated from our database without $canSplit$ and $isSplit$. The set correctly classify 83% of the entries in the database used. This heuristic has a fitness of -0.0234.

its coverage dependent on the order of the conditions found, rearranging their positions would change the outcome. We believe that having a heuristic in which the statements are presented in descending order of their positive impact on the gameplay would lead to a more flexible heuristic, i.e. if the number of statements in the heuristic is too overwhelming for a novice player, we could discard the bottom ones and have a tighter set size with effective fitness value. We also wanted to test the hypothesis of whether a heuristic generated in such fashion could outperform the decision list generated by RIPPER using a smaller number of statements, since we know that playing the Basic Strategy was more profitable.

VI. EXHAUSTIVE-GREEDY SEARCH

To produce a heuristic with the most impactful statement in each step we decided to use a exhaustive-greedy search algorithm. After playing the game for each of the possible statements, it picks the one that achieves the best average score. The algorithm starts with just the default action and with each new iteration the candidate statements are tested at the bottom of the current set, before the default move. Each selected statement is appended right before the default action. We wanted to create a set of statements that could be flexible, so that we can reduce the complexity by trimming the statements on the heuristic while trying to have low impact on its fitness. We then append the rules to enabling removing the later statements without impacting the previous.

The algorithm tries different default actions when searching for a heuristic. Having *double down* or *split* as default moves did not generate statements with good fitness as they were a much smaller part of the strategy when compared to the two other moves. This lead to a lengthier set, as statements are added to cover both *hit* and *stand*. The 1 statement heuristic found proved *hit* to be a better default action for the algorithm we were using. Figure 5 shows the same 1 statement heuristic found with *hit* and *stand* as the default action.

```
if $16 \leq P_{pts}$ then STAND
else HIT
```

```
if $P_{pts} \leq 15$ then HIT
else STAND
```

Fig. 5. The first statement found using exhaustive-greedy search with *hit* and *stand* as default moves. Fitness is -0.0546.

Since the exhaustive-greedy search algorithm added statements to the end of the set, the first statement found had a great impact. The 1 step heuristics found picked the same moves in every setup, but the space left for the next ones to be found is very different. The *stand* default move set could only grow new statements for when the player had more than 15 points, which is a much more limited space than the one where player has less than 16 points. And when looking at the Basic Strategy we can see that the higher hand value strategies are composed mostly of *stand* since the chances of *busting* are very high. With such conditions, we decided to keep *hit* as the default action and reduce the search space that the algorithm had to explore. Since RIPPER also has the same default move, it is also easier to compare the heuristics generated.

To run the algorithm we needed to generate all possible statements to evaluate. When generating them, we can apply domain knowledge to reduce the search space: $canSplit$ only needs to be called if the move being targeted is *split* and we don't need to constraint the player points by odd numbers in that scenario. Generating all statements to cover all possible combinations of conditions and moves, we have 9,405 statements for *stand*, 9,405 for *double down* and 3,025 for *split*, for a total of 21,835 different statements. Because the search space is already this large, we decided not to generate heuristics that use $isSoft$, as it would result in doubling the number of possible statements, for what it seemed to be a small gain in fitness. Figure 6 shows the heuristic found.

```
if $16 \leq P_{pts}$ then STAND
else if $9 \leq P_{pts} \leq 11$ and $D_{pts} \leq 8$ then DOUBLEDOWN
else if $13 \leq P_{pts} \leq 15$ and $D_{pts} \leq 6$ then STAND
else HIT
```

Fig. 6. Greedy search selecting the highest average scoring statement. The algorithm reached a local maximum on this 3 step heuristic. Fitness is -0.0302.

When comparing both, the RIPPER simpler set, shown on Figure 4, outperforms the exhaustive-greedy search by 0.007 average fitness score. That result comes from increasing complexity: the RIPPER heuristic has 3 more statements.

Greedy search for the best heuristic revealed an interesting aspect of the game: if we had a single step to follow it would be for the player to *stand* when having 16 points or more. That contradicts a first impression from looking at the Basic Strategy that the player should *stand* on 17 or more and on 16 if the dealer has 6 or less points, *hit* otherwise. Following the nature of our decision structure, since we are only looking at a 1-statement heuristic, what the greedy selection shows is that, if we look at only the player points, when having to make a choice of which move to make at 16 points, in average, is better to *stand* than to *hit*. This is specially interesting since 16 is considered the worst hand in Blackjack [34].

Another interesting result was how fast the local optimum was reached. The contender for fourth statement had no impact since its condition was already covered in the set. As a result of such and the observation that a heuristic that would more closely resemble the Basic Strategy could have a better average, we believe that greedily searching for a heuristic is very prone to getting stuck on local optimums. In order to get better results from searching we decided to expand our greedy approach.

VII. Expanding the exhaustive-greedy search

Keeping the exhaustive search approach, we try to generate a new heuristic by making the greedy aspect more expansive. For that, we decided to no longer pick only the best statement in each step, but instead register the top 5 ranking statements for each step. On every iteration of the algorithm works in the same fashion as exhaustive-greedy search, but now the 5 top results are store and iterated upon separately. After the last iteration we take the set with the best average score in the last step. Figure 7 shows the heuristic found.

```
**if** $17 \leq P_{pts}$ **then** STAND
**else if** $13 \leq P_{pts}$ **and** $D_{pts} \leq 6$ **then** STAND
**else if** $10 \leq P_{pts} \leq 11$ **and** $3 \leq D_{pts} \leq 9$ **then** DOUBLE
**else** HIT
```

Fig. 7. Using a greedy exhaustive search and considering the top 5 average scoring statements. The top resulting heuristic the algorithm found is shown. This set is a local maximum. This heuristic has a fitness of -0.0247.

Searching over the 5 best result shows that the heuristic found by the greedy algorithm was a local optimum, despite also being subject to such. The heuristic found outperforms and is as simple as the one found by exhaustive-greedy, and is also closer to the Basic Strategy. The first two statements found are exactly on par with the Basic Strategy, but looking at the bottom, we can see that, to match the Basic Strategy, it needed to also include *double down* when the dealer is showing a 2. That result comes from the fact that the case of the dealer showing a 2 against a player's hand of 10 or 11 is present in only a very small number of games, if at all.

Another downside is that the new heuristic found was another local optimum. Running another loop of the algorithm finds no new statements, at the top 5, that cover a new scenario. We can see from looking at the Basic Strategy layout that there are other cases the set does not cover.

A better search algorithm would be needed to look for the best n-statement heuristic algorithm. By limiting the size of our search space by removing $isSoft$ there is still a very large search space. An exhaustive search of the full space would guarantee the optimum heuristic, but would also be very computationally expensive, even by pruning the statements that do not further cover new elements. To look further for a better strategy for generating simple heuristics we then turn to algorithms that employ different search strategies.

VIII. Axis-Aligned Search

Another way to search the space of possible heuristics is to use an axis-aligned search algorithm, modifying each parameter in the conditions individually along each single dimension. For example, for a 1-condition heuristic, we would individually test all lower bounds $\beta_{lower}$for P_{pts} keeping all other values fixed, then all upper bounds β_{upper} keeping all other values fixed, etc. also exploring the lower and upper bounds for D_{pts} and all actions for the condition and default action. Because we do not have a large number of possible values for each parameter (between 3 and 21 depending on the parameter), we can search all values along each single dimension – on the order of around 100 different heuristics to test for a 1-condition Blackjack heuristic. However, we don't need to evaluate all of these heuristics because many of them are equivalent, so we first pass through and remove any heuristics which are equivalent to any others. This has the effect of only needing to search approximately 75 statements per iteration for a 1-condition heuristic, approximately 210 for a 3-condition heuristic, and approximately 375 statements per iteration for a 5-condition heuristic. This axis-aligned search has the benefit that it can get out of local maximum as long as another maximum exists anywhere on one of the dimensions holding all other dimensions fixed. For games where parameters may be too many values to search completely, one could use importance sampling or local neighborhood search to find a smaller number of parameters to choose along each dimension.

The algorithm proceeds as follows. We begin from a random statement, and find its fitness. We then find all unique statements along each individual dimension and find their fitnesses. We take the one with the best fitness and repeat the algorithm from there. When we find a statement that has the best fitness of any of its candidates, we terminate and return the best candidate found, which is a local best but not necessarily a global best. Because this is not guaranteed to find the global best (as is also the case for simulated annealing and other optimization algorithms), we repeat the process several times and return the best from all runs. We find that axis-aligned search performs very well, finding some of the highest fitness heuristics for a given complexity, but it has a high variance and often finds badly performing ones. It is therefore essential to run several times. It can also be used as a final touch-up process for other algorithms to search for any final improvements.

For 1-condition heuristics, axis-aligned search would very often find the optimal heuristics presented in Figure 5. The best performing 3-condition heuristic is presented in Figure 8. Split/Hit refers to the player using the split action when legal, otherwise they would treat this as a hit. No other algorithm found this highly performing heuristic, with a high fitness of -0.0221 especially given its relatively low complexity.

```
**if** $17 \leq P_{pts}$ **then** STAND
**else if** $13 \leq P_{pts}$ **and** $D_{pts} \leq 6$ **then** STAND
**else if** $10 \leq P_{pts} \leq 11$ **and** $D_{pts} \leq 9$ **then** DOUBLEDOWN
**else** SPLIT/HIT
```

Fig. 8. Starting from random conditions, the best 3-condition heuristic observed for multiple runs of the Axis-Aligned Search algorithm. The heuristic has a fitness of -0.0221.

IX. Genetic programming

Finally, we examined using genetic programming [35] to find the best decision list strategies at a given complexity for use by novice players. Our approach is similar to that used in [25]. We used the DEAP [36] framework for genetic algorithms (GA), which includes implementations for various types of stochastic optimization. We experimented with a basic genetic algorithm as well as $(\mu + \lambda) - ES$ [37].

We test the population of potential strategies by simulating 13*13*13*500 = 1,098,500 games for each individual. The genotype contains the default action and a list of conditions bounds and actions. Each simple condition is made of four possible clauses $\pi_{lower} \leq P_{pts}$, $P_{pts} \leq \pi_{upper}$, $\delta_{lower} \leq D_{pts}$, and $D_{pts} \leq \delta_{upper}$ as described in Section III. Because a clause is meaningless if $\pi_{upper} < \pi_{lower}$ or $\delta_{upper} < \delta_{lower}$, each condition instead stores 4 positive integers $\pi_{lower}, \pi_{upper} - \pi_{lower}, \delta_{lower}, \delta_{upper} - \delta_{lower}$ to ensure that upper bounds are never smaller than lower bounds. During mutation, we ensure that the ranges are not violated by clamping to sensible values that can only occur in Blackjack. We also include a boolean for each clause, that allows the clause to be easily turned on and off during mutation. For complex strategies that are allowed to include $canSplit$ and $isSoft$ this adds 3 more variants where each item can be $=$, $\neq$, or *don't care*.

The hyperparameters used for controlling the genetic algorithm needed to be tuned depending on the complexity of the statements that were to be generated. For heuristics with 1 to 3 conditions, we found good results using 30% chance of modifying each value in a genotype, 20% chance to flip on/off a clause, 20% chance to shuffle the order of condition/actions, 40% chance to create an offspring with mutation, and 20% chance to create an offspring with crossover, a population size of 100, and 30-50 generations. For heuristics with 5 or more conditions, we found better results by using a 10% chance of modifying values, leaving all clauses on, and not shuffling the clauses, and switching to a $(\mu + \lambda) - ES$ framework with $\mu = 20$ and $\lambda = 200$ for 100 generations.

Our GP did not find as highly optimized heuristics as the axis-aligned method, but on most runs would find something adequate. In general, GP seemed to have a smaller variance in final fitness but did not have the overall best results for a given complexity. However, we believe that with more generations, larger populations, and more highly-tuned hyperparameters the GA would be likely to perform better.

X. Discussion

We used several methods to generate heuristics for Blackjack. The results show expressions of different lengths, heuristic complexity and fitness. The different methods also have different run-time computational complexity. The RIPPER algorithm is very fast since the database only had 550 entries, but relies on an existing database of optimal moves. Generating the database to cross check the Basic Strategy was costly, but needed to be done only once. Doing exhaustive-greedy search was costly due to the amount of possible statements, and our top 5 greedy variation was even worse, being exponentially more expensive for generating more complex heuristics. Genetic Programming running time is related to the size of the population and number of generations needed to converge to a good solution, and requires tuning of hyperparameters. Meanwhile Axis-Aligned Search searches a much smaller number of statements, compared to the exhaustive-greedy searches, but often finds bad performing heuristics given a bad random starting point. Axis-aligned search will likely pose problems for games with significantly more parameter space to search without some tuning.

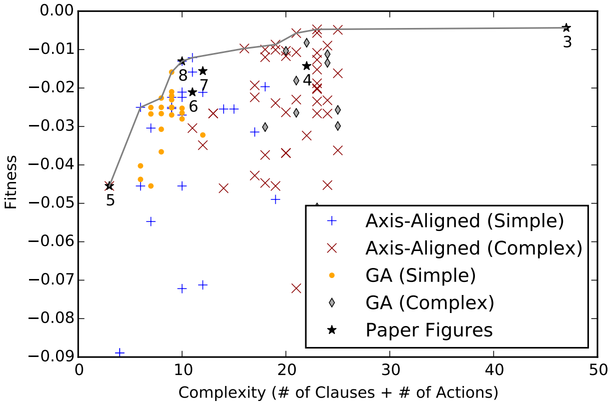


Fig. 9. Comparison of heuristic complexity vs fitness for various heuristics. The numbers in the graph refer to the corresponding Figure index for the different heuristics shown throughout this paper. 3 is RIPPER (complex), 4 is RIPPER (simple), 5 is 1-rule exhaustive-greedy, 6 is 3-rule exhaustive-greedy, 7 is top-5 greedy and 8 is Axis-Aligned.

The results show an interesting relationship between the heuristics complexity and fitness. Figure 9 shows the comparison of different runs of each of our algorithms, plotting the complexity of the heuristic against its fitness. *Simple* heuristics do not use the $isSoft$ or $canSplit$ clauses, and *complex* heuristics use all possible clauses. This shows that low-complexity rules can have a large improvement in fitness, but then smaller gains in fitness require larger increases in complexity. RIPPER expressions show up as outliers in terms of complexity, having twice as many causes and actions as most of the expressions found, but also have the heuristic with the highest ranked fitness. The highest fitted Axis-Aligned complex heuristics are half as complex as the top fitted RIPPER, but have very close fitness. By analyzing the hull atop the fitted results over the increasing complexity, we can see the trend in fitness gain is much greater on the lower complexity results.

We can observe the impact of increasing the space of possible clauses by introducing of $canSplit$ and $isSoft$ leads to a small fitness gain, but a much higher increase in complexity. The best heuristics generated with their addition range from 2 to close to 5 times more complexity.

We believe that our framework can be extended to other games for generating beginning heuristics, but certainly with some significant challenges. One of the major challenges is finding a primitive set of operations to include in the conditions and actions. For blackjack, these primitives are relatively obvious and easy to implement (perhaps because we have the Basic Strategy to refer to) but in other games, even simple ones like Tic-Tac-Toe, the conditions and actions can become far more complicated to encode.

For games that have a distinction between tactics and strategy, we predict that different trees, conditions, and actions would be required to represent differences between short-term tactical moves and long term high-level planning. Perhaps a player would need to first use a FFT heuristic to figure out a high-level goal to reach, and then another FFT to implement that goal.

Some games have an existing known good strategy, either obtained through collective wisdom gathered after collectively many hours, years, or centuries of study and play, or by game tree exploration using minimax or Monte Carlo Tree Search. In this case, using a simplification algorithm such as RIPPER can make sense. For others, when the game is newly designed and doesn't yet have known-good strategies [38] or the output of a computational creativity game generation system [39], strategies need to be developed from the ground up, as we do with the axis-aligned search and genetic algorithm methods.

We plan in the future to confirm that our heuristics are indeed easy to learn and execute by novice players, by comparing accuracy and speed at which novices can perform the correct Blackjack plays given a fixed set of cards. Such a study can also give us a more accurate complexity measure for clauses.

XI. Conclusions and Future Work

This paper poses the problem of algorithmically generating compact heuristics that can be easily learned by novice human players. As an example we used Blackjack, a simple game for which the optimal strategy is already known but requires a significant amount of time to learn. We explored four different approaches to discovering simple fast and frugal heuristics for novice-level Blackjack. It was found that inducing conditions from a known strategy table with the RIPPER algorithm resulted in heuristics that were too large and not much better than much smaller heuristics. Exhaustive search for expressions that were linked together with greedy search was found to be comparatively slow and suffer from the tendency to find globally suboptimal choices inherent in the greedy search. Genetic programming and axis-aligned search were both able to find the desired compact yet effective heuristics. While we have shown that finding these heuristics is doable for Blackjack, the challenge of scaling up to games with larger state and action spaces where optimal play is not already known remains. We hypothesize that solving this will involve automatic extraction of relevant behavioral and positional primitives.

References

[1] H. A. Simon, "Theories of bounded rationality," *Decision and organization*, vol. 1, no. 1, pp. 161–176, 1972.

[2] D. Kahneman, "Maps of bounded rationality: Psychology for behavioral economics," *The American economic review*, vol. 93, no. 5, pp. 1449–1475, 2003.

[3] G. Gigerenzer and D. G. Goldstein, "Reasoning the fast and frugal way: models of bounded rationality." *Psychological review*, vol. 103, no. 4, p. 650, 1996.

[4] J. P. Gee, "Learning by design: Good video games as learning machines," *E-Learning and Digital Media*, vol. 2, no. 1, pp. 5–16, 2005.

[5] G. S. Elias, R. Garfield, K. R. Gutschera, and P. Whitley, *Characteristics of games*. MIT Press, 2012.

[6] G. Gigerenzer, "Fast and frugal heuristics: The tools of bounded rationality," *Blackwell handbook of judgment and decision making*, pp. 62–88, 2004.

[7] R. L. Rivest, "Learning decision lists," *Machine learning*, vol. 2, no. 3, pp. 229–246, 1987.

[8] D. Ashlock, M. Joenks, J. R. Koza, and W. Banzhaf, "Isac lists, a different representation for program," in *Genetic Programming 1998*. Morgan Kaufmann, 1998, pp. 3–10.

[9] E. O. Thorp, *Beat the dealer: A winning strategy for the game of twenty-one*. Vintage, 1966.

[10] M. Campbell, A. J. Hoane, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1, pp. 57–83, 2002.

[11] M. Buro, "From simple features to sophisticated evaluation functions," in *Computers and Games*. Springer, 1998, pp. 126–145.

[12] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron, "A world championship caliber checkers program," *Artificial Intelligence*, vol. 53, no. 2, pp. 273–289, 1992.

[13] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[14] A. Hauptman and M. Sipper, *GP-endchess: Using genetic programming to evolve chess endgame players*. Springer, 2005.

[15] A. Benbassat and M. Sipper, "Evolving lose-checkers players using genetic programming," in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*. IEEE, 2010, pp. 30–37.

[16] Y. Azaria and M. Sipper, "Gp-gammon: Genetically programming backgammon players," *Genetic Programming and Evolvable Machines*, vol. 6, no. 3, pp. 283–300, 2005.

[17] L. Barone and L. While, "An adaptive learning model for simplified poker using evolutionary algorithms," in *Congress on Evolutionary Computation, CEC 99*, vol. 1. IEEE, 1999.

[18] B. Vowk, A. S. Wait, and C. Schmidt, "An evolutionary approach generates human competitive corewar programs," in *Workshop and Tutorial Proceedings Ninth International Conference on the Simulation and Synthesis of Living Systems (Alife XI)*. Citeseer, 2004, pp. 33–36.

[19] Y. Shichel, E. Ziserman, and M. Sipper, "Gp-robocode: Using genetic programming to evolve robocode players," in *8th European Conference on Genetic Programming*, vol. 3447, 2005, pp. 143–154.

[20] J. Togelius, S. Karakovskiy, J. Koutník, and J. Schmidhuber, "Super mario evolution," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*. IEEE, 2009, pp. 156–161.

[21] W. B. Langdon and R. Poll, "Evolutionary solo pong players," in *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, vol. 3. IEEE, 2005, pp. 2621–2628.

[22] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.

[23] A. Agapitos, J. Togelius, and S. M. Lucas, "Evolving controllers for simulated car racing using object oriented genetic programming," in *Genetic and evolutionary computation*. ACM, 2007, pp. 1543–1550.

[24] D. B. Fogel, "Evolving strategies in blackjack," in *Congress on Evolutionary Computation, CEC2004*, vol. 2, 2004, pp. 1427–1434.

[25] M. Jelev and S. Koch, "Genetic blackjack," 2009.

[26] A. Pérez-Uribe and E. Sanchez, "Blackjack as a test bed for learning strategies in neural networks," in *Neural Networks Proceedings, 1998. IEEE World Congress on Computational Intelligence. The 1998 IEEE International Joint Conference on*, vol. 3. IEEE, 1998, pp. 2022–2027.

[27] G. Kendall and C. Smith, "The evolution of blackjack strategies," in *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, vol. 4. IEEE, 2003, pp. 2474–2481.

[28] M. R. Schiller and F. R. Gobet, "A comparison between cognitive and ai models of blackjack strategy learning," in *KI 2012: Advances in Artificial Intelligence*. Springer, 2012, pp. 143–155.

[29] F. Gobet, J. Retschitzki, and A. de Voogt, *Moves in mind: The psychology of board games*. Psychology Press, 2004.

[30] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and R. Qu, "Hyper-heuristics: A survey of the state of the art," *Journal of the Operational Research Society*, vol. 64, no. 12, pp. 1695–1724, 2013.

[31] L. Humble, *The World Greatest Blackjack Book*. Main St. Books, 2010.

[32] R. R. Baldwin, W. E. Cantey, H. Maisel, and J. P. McDermott, "The optimum strategy in blackjack," *Journal of the American Statistical Association*, vol. 51, no. 275, pp. 429–439, 1956.

[33] W. W. Cohen, "Fast effective rule induction," in *Intl Conference on Machine Learning*, 1995, pp. 115–123.

[34] L. Revere, *Playing Blackjack as a Business: A Proffessional Player's Approach to the Game of" 21"*. Lyle Stuart, 2000.

[35] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, *A field guide to genetic programming*. Lulu. com, 2008.

[36] D. Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, C. Gagné *et al.*, "Deap: A python framework for evolutionary algorithms," in *Genetic and evolutionary computation*. ACM, 2012, pp. 85–92.

[37] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies–a comprehensive introduction," *Natural computing*, vol. 1, no. 1, pp. 3–52, 2002.

[38] A. Isaksen, M. Ismail, S. J. Brams, and A. Nealen, "Catch-up: A game in which the lead alternates," *Game & Puzzle Design*, vol. 1, no. 2, 2015.

[39] C. Browne and F. Maire, "Evolutionary game design," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 2, no. 1, pp. 1–16, 2010.