Journal of Artificial Intelligence Research 68 (2020) 447-461 Submitted 12/2018; published 06/2020

Automated Conjecturing II: Chomp and Reasoned Game Play

A. Bradford
J. K. Day
L. Hutchinson
B. Kaperick
C. E. Larson CLARSON@VCU.EDU
M. Mills
D. Muncy
Department of Mathematics and Applied Mathematics
Virginia Commonwealth University
Richmond, VA 23284, USA

N. Van Cleemput NICO.VANCLEEMPUT@GMAIL.COM
Department of Applied Mathematics, Computer Science and Statistics
Ghent University
9000 Ghent, Belgium

Abstract

We demonstrate the use of a program that generates conjectures about positions of the combinatorial game Chomp—*explanations* of why certain moves are bad. These could be used in the design of a Chomp-playing program that gives reasons for its moves. We prove one of these Chomp conjectures—demonstrating that our conjecturing program can produce genuine Chomp *knowledge*.

The conjectures are generated by a general purpose conjecturing program that was previously and successfully used to generate mathematical conjectures. Our program is initialized with Chomp invariants and example game boards—the conjectures take the form of invariant-relation statements interpreted to be true for all board positions of a certain kind. The conjectures describe a theory of Chomp positions.

The program uses limited, natural input and suggests how theories generated on-the-fly might be used in a variety of situations where decisions—based on reasons—are required.

1. Introduction

We describe a program that produces a theory of positions of the combinatorial game Chomp. The theory consists of statements that have a truth value. We include a *proof* of one of these generated statements as a demonstration of the potential high-quality of the program's output.

The *input* of our CONJECTURING program are a very small number of examples of Chomp positions. The *output* will be a collection of human-comprehendable (that is, relatively short) statements that describe certain desired board positions; these can, in turn, be used as a basis of move selection. The theory is produced by a general-purpose automated conjecturing program. The main purpose of this note is to demonstrate how automated con-

jecturing programs might be used in the design of programs that make *reasoned, explainable* actions for Chomp-like games.

An essentially unlimited number of Chomp *invariants* (numbers independent of Chomp position representation) can be defined. These invariants together with labeled Chomp positions constitute the knowledge given to the program. The conjectures produced by our program naturally provide explanations for a version of one-step lookahead. This model suggests a theory for how a program can learn to play Chomp-like games well.

The underlying idea of our program is general and not Chomp-specific. It does not use knowledge obtained from experts, nor does it use traditional machine learning approaches (neural nets, support vector machines, etc.) or big data. The idea is to make conjectures based on very limited experience, and use new information to revise and improve its conjectures. (This is what mathematicians do and may be a more general cognitive strategy). A program might then be designed to make moves using those conjectures, possibly in conjunction with other heuristics or machine learning techniques.

2. Background & Previous Work

2.1 Automated Discovery

Research on automated mathematical discovery was conceived by Turing (1948) and there is current work in a variety of areas (Larson & Van Cleemput, 2016; Larson, 2005). The first programs to prove theorems were developed in the 1950s (Simon & Newell, 1958); and the 1996 computer proof of the Robbins Conjecture (McCune, 1997) was a milestone in this area. Fajtlowicz's Graffiti was the first program to produce conjectures that led to new theorems, published in research journals (new mathematical *knowledge*). Since Graffiti produces statements that are relations between mathematical invariants, it is worth noting related programs. The best-known research on discovering scientific laws that are simple relations between invariants (measurements) is the work of Bacon and its successors, comprehensively described by Langely, Simon, Bradshaw, and Zytkow (1987); these programs reproduced known physical laws, rather than conjecture or discover any new laws. (Similarly Lenat's often-cited AM reproduced existing mathematical conjectures and theorems—rather than made any new mathematical contribution (Lenat, 1977)). The program of Todorovski, Džeroski, and Kompare (1998) discovered relations between measurements in a novel real-world situation; their example involves very few data points and very few variables, and does not seem to have led to generalizations. The work of Schmidt and Lipson (2009) is an impressive experiment along similar lines; their program discovered conservation laws for specific real-world physical systems.

Epstein's FORR (FOr the Right Reasons) scheme and `Hoyle` program (Epstein, 1994, 2001; Ratterman & Epstein, 1995) share goals with our research: these shared goals include game playing programs that use limited memory, limited game tree search, giving reasons and learning from experience. In both cases, limited memory usage is due to a use of a limited number of examples. Limited game tree search for `Hoyle` is by design: the reasons built-in to Hoyle include valuing moves with 1- or 2-ply payoffs. A program using conjectures as we describe here can be designed to only look at the current available moves. Both of our programs are designed to give reasons: in the FORR paradigm, some of these must be provided by experts—then the program's algorithm decides how much weight to give

each of these provided reasons. Our idea is not to use any expertly-provided reasons at all—simply to see what can be deduced from known winning and losing positions. With FORR weights are revised as relevant examples are adduced—with CONJECTURING new examples may lead to falsified conjectures—that can never again be produced. In these senses new examples provide new experiences, and these programs are designed to leverage them.

The example statements produced by the CONJECTURING program will be necessary conditions for winning positions for the current player; these are the positions the current player should attempt to attain on her move—and can—if she currently has a winning position. So this research might be seen as a contribution to *explainable artificial intelligence* (XAI): these simple statements might be taken as *explanations* of a Chomp player's actions. This area of research is new, and its goals are still being clarified. The main idea is that artificial intelligence programs using, for instance, neural nets have output that would be formally described as a very complicated function—with a large number of variables and weights. In contrast, many scientific laws (Newton's Laws, Boyles Law, etc) used and accepted as explanations express relationships between a small number of variables. There is no accepted dividing line separating statements that might be counted as explanations from more complicated (or less explanatory) ones. There are a variety of goals here—possibly including the desire to have statements that could plausibly be proved from other accepted laws or statements. Conflicting interests and standards of success will eventually shake out. The statements produced here as examples have relatively simple form, have few variables, and coefficients that are small integers or simple functions of them.

2.2 Chomp

Consider, for example, Fisher's seminal example of automating iris classification by developing a linear discriminant from feature data (Fisher, 1936). Here he uses a data set consisting of flower samples of three iris species and four measurements (or *characters* or *features*) from each sample. His goal was to find a function which, when given features as input, correctly produces the iris species-type as output. He starts with species *Iris setosa* and *Iris versicolor* and features x_1, x_2, x_3 and x_4 for, respectively, the flowers' sepal length, sepal width, petal length and petal width, and 50 samples of each flower and looks for a discriminating function of the form:

$$X = \lambda_1 d_1 + \lambda_2 d_2 + \lambda_3 d_3 + \lambda_4 d_4,$$

where d_i is the difference of the mean of the feature from x_i. Fisher ultimately derives the function,

$$X = d_1 + 5.9037 d_2 + 7.1299 d_3 + 10.1036 d_4,$$

where positive values of X will identify *setosa* flowers and negative values of X will identify *versicolor* flowers. In this case it seems unlikely that this classifier might be proved from simpler (biological) relations or from analysis of the involved concepts. In contrast our program produces simple relations that might be easier to investigate (and we give an example of one that is *proved* from the involved concepts).

Our research extends the authors' program CONJECTURING, that produces invariant-relation conjectures, and is based on a heuristic of Fajtlowicz (1995). CONJECTURING and

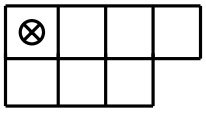

Figure 1: A 2-row Chomp position `[4,3]` with the poison cookie in the upper left corner.

related experiments are described by Larson and Van Cleemput (2016). The user of this program may input example objects of any type, choose invariants (real numbers that can be computed from the objects, specified as functions) that may appear in the conjecture statements, choose a specific invariant that will appear on the left-hand side of the conjecture, and choose the form of the inequality: either upper bounds or lower bounds for the chosen invariant. The reported conjectures came from the domains of graph theory, matrix theory, number theory, and combinatorial game theory. More recently we have demonstrated the utility of our program for making conjectures in mathematical chemistry (Hutchinson, Kamat, Larson, Mehta, Muncy, & Van Cleemput, 2018), several of which we proved. Our program is open-source, and operates in Sage (a free and growing mathematical computing environment, similar to Maple, Matlab and Mathematica). The program, examples, and set-up instructions are available at: `http://nvcleemp.github.io/conjecturing/`

The program produces conjectures, in the form of inequalities between algebraic relations of the input invariants, that are true for all example objects that are provided to the program. By the design of the program, each produced conjecture is significant with respect to the previously produced conjectures—in the sense that each conjecture gives a better bound for one of the example objects than any previously produced conjecture (this condition defines our use of the term "significant"). This also implies that each newly produced conjecture is not implied by the previously produced conjectures. Furthermore, it means that the number of conjectures (of any particular form) cannot exceed the number of example objects.

Chomp is a two-player perfect information impartial game. It was invented by David Gale in the 1970s (Gale, 1974). We chose this game for our initial experiments because of its simplicity. The game begins with a full rectangle of cookies; this rectangle consists of an arbitrary number of rows and columns. Each player alternately removes (or eats) a cookie from the board. In our version, when a cookie is removed, all cookies to the right of this cookie and all cookies below any of these are removed. The upper left-most cookie is the poison cookie. The player to remove the poison cookie loses. Passing is not allowed: a player must chose a cookie to remove or eat while cookies remain. Fig. 1 illustrates a two-row Chomp board position. The circle with the cross represents the poison cookie, while empty squares represent the remaining cookies.

Gale proved that the first player has a winning strategy: that is, with perfect play, the first player can always win. Nevertheless, on even relatively small game boards it is computationally impossible to determine this winning strategy. Successful play depends on theory and experience. Winning strategies for specific positions are known. For instance, one row and two row Chomp are solved: it can be efficiently determined whether any one or two row position is a winning position (Zeilberger, 2001). *Three row Chomp is unsolved.* While some results are known about three row positions, Doron Zeilberger,

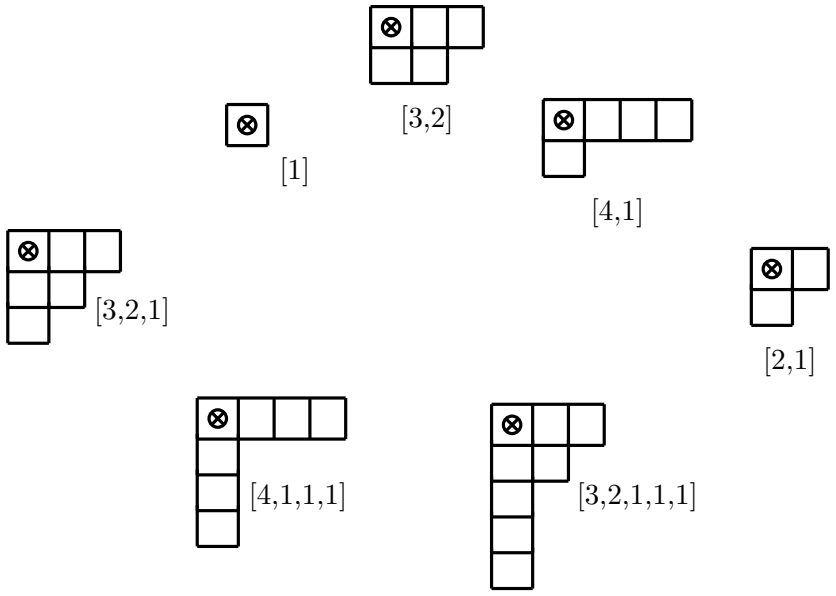

Figure 2: Examples of Chomp board positions and corresponding list representations.

a noted mathematician, proponent of experimental mathematics, Chomp enthusiast and researcher, believes that the three row case may be the borderline between easy-to-solve and intractable positions (Zeilberger, 2001).

Game boards can be represented in a variety of ways. We represent them with a non-increasing sequence of positive numbers: the first number represents the number of cookies in the first row, the nth number represents the number of cookies in the nth row (square brackets are used here as this is Python's *list* notation—how these are represented in our code—and code snippets below). The rules of the game guarantee that the sequence is non-increasing. The notation `[3]`, for instance, represents a single-row Chomp board with three cookies; other examples may be found in Fig. 2. An example of a complete game would be the following. The players may play beginning with the rectangular 2×3 board `[3,3]`. The first player might then choose the lower rightmost cookie. Since no other cookie is below or to the right, the resulting board is `[3,2]`. If the second player then takes the second cookie in the first row, together with the single cookie below it and the single cookie to its right, the resulting board is `[1,1]`. The first player will now take the single remaining non-poison cookie, leaving the board `[1]`. Then the second player is forced to take the poison cookie and loses.

Research on machine learning does, of course, lead to conjectured relationships between variables that are in turn used to make predictions of one or more variables in terms of others. A trained neural net, for instance, can be viewed as a black box representing a function that produces an output for every input in its domain. These functions are complex and of a different character than, say, classical scientific laws: in particular, there is less hope of deriving these functions or relationships from simpler existing laws.

3. The Dalmatian Heuristic

Fajtlowicz's *Dalmatian* heuristic is used to conjecture relations between real number invariants of objects. For Chomp the objects are board positions and the invariants are any numbers that can be computed from these positions. This Dalmatian heuristic comprises a truth-test and a significance test.

Both Graffiti and Conjecturing produce inequalities between algebraic relations of the input invariants. These are then checked to be true for all examples that are provided to the program. This is the *truth test*. If a produced statement is false for an input object, the statement is rejected as a potential conjecture. Each statement is then tested for *significance* with respect to the input objects and the database of previously produced conjectures. Informally, a statement is "significant" if it is not implied by the totality of previously made conjectures. By the design of the program, each produced conjecture is then significant with respect to the previously produced conjectures.

In many well-studied research areas (such as chess, Go, or mathematical fields) there are a large number of previously defined invariants. For well-studied games, these would include position-evaluation functions—these *are* game position invariants. With other less-studied games, including Chomp, there is no existing foundation to build on: an initial attempt must be made to define potentially useful position invariants. These could be modified, added to and supplanted as more researchers think about a game.

The following Chomp invariants were created for the sole purpose of having a large collection of numbers that, together, would capture a large amount of information about a board position. The utility of a particular invariant cannot be known until after conjectures are generated. Many other Chomp invariants are conceivable and, possibly, more natural, useful, or informative. The invariants that we used included:

`number_of_rows`	the number of non-empty rows,
`number_of_columns`	the number of non-empty columns
`number_of_cookies`	the number of cookies remaining on the board,
`full_rectangle`	the product of the number of rows and columns,
`rows_of_different_length`	the number of rows with different numbers of cookies,
`rank_ratio`	the number of rows of different length divided by the number of rows,
`squareness`	the absolute value of the difference between the number of rows and the number of columns,
`row_product`	the product of the number of cookies in each row,
`column_product`	the product of the number of cookies in each column,
`average_cookies_per_column`	the average number of cookies in the columns,
`average_cookies_per_row`	the average number of cookies in the rows
`duplicate_rows`	the number of rows minus the number of rows with different numbers of cookies,
`duplicate_columns`	number of columns minus number of columns of different length,

`smallest_row_size`	fewest number of cookies in any row,
`smallest_column_size`	fewest number of cookies in any column,
`largest_row_size`	most number of cookies in any row,
`largest_column_size`	most number of cookies in any column,
`cookies_inside_ratio`	the number of cookies that are not on the top row or leftmost column divided by the number of cookies.

Computers can easily exhaustively generate and evaluate all expressions formed from standard mathematical ingredients for relatively small numbers of example mathematical objects. These expressions—and their corresponding values for example sets of objects—can then be utilized in a variety of ways. Consider the search for upper bounds for some invariant α. For these purposes we take an *invariant* to be a real-valued function of a Chomp position. Let β_1, β_2, ..., β_k be Chomp invariants. An upper bound β of α will be some mathematical function of the β_i's. This function will involve arithmetic operations, algebraic operations, or any other mathematical operations. Some real-number operators would include addition, square root, and taking the minimum of two numbers ($+$, $\sqrt{}$ and min). In this case (and if $k = 3$ for example) the complexity-1 expressions would be the invariants themselves: $\beta_1, \beta_2, \beta_3$. The complexity-2 expressions would be all the expressions that can be formed from a mix of two operators or invariants. Since $\sqrt{}$ is the only unary operator, the only possibilities are: $\sqrt{\beta_1}$, $\sqrt{\beta_2}$, $\sqrt{\beta_3}$. The complexity-3 expressions are: $\beta_1 + \beta_2$, $\beta_1 + \beta_3$, $\beta_2 + \beta_3$, $\beta_1 + \beta_1$, $\beta_2 + \beta_2$, $\beta_3 + \beta_3$, $\min(\beta_1, \beta_2)$, $\min(\beta_1, \beta_3)$, $\min(\beta_2, \beta_3)$, $\min(\beta_1, \beta_1)$, $\min(\beta_2, \beta_2)$, $\min(\beta_3, \beta_3)$, $\sqrt{\sqrt{\beta_1}}$, $\sqrt{\sqrt{\beta_2}}$, and $\sqrt{\sqrt{\beta_3}}$ (a modern computer algebra system can identify and remove expressions equivalent due to additive commutativity, etc). A program can recursively generate all possible β's up to any specified complexity. Generating expressions will face combinatorial explosion—but there is no difficulty in generating all (relatively small) human-comprehendible expressions. (Our C-language expression generator can generate more than 100 million expressions per second, depending on the complexity of the expressions, on a standard 2018 laptop—with 8 GB RAM and a 2.6 GHz core). Our CONJECTURING program can evaluate these expressions on the fly for a particular Chomp position. These generated, evaluated expressions are the main ingredients in generating conjectures.

Here is a formal description of the Dalmatian heuristic. Let $B_1, \ldots, B_n$ be Chomp positions (boards). The generated inequalities can be interpreted as being true for *all* Chomp positions (in general, or of any input Chomp position type). The inequality $\alpha \leq \beta_1 + \beta_2$ can be interpreted, for instance, as, "For every Chomp position B, $\alpha(B) \leq \beta_1(B) + \beta_2(B)$." A conjectured upper bound β is only added to the database of conjectures if the bound passes the following two tests.

1. (*Truth test*) The candidate conjecture $\alpha \leq \beta$ is true for all of the stored objects $B_1, \ldots, B_n$, and

2. (*Significance test*) There is an object $B \in \{B_1, \ldots, B_n\}$ such that $\beta(B) < \min\{c_1(B), \ldots, c_r(B)\}$, where $c_1, \ldots, c_r$ are the currently stored conjectures. That is, the candidate conjecture would give a better bound for $\alpha(B)$ than any previously conjectured (upper) bound.

Here is a simple example. We designed a Python class `ChompBoard` that takes board position representations as input and returns an *object* that has invariants as the objects' *methods*.

```
objects = [ChompBoard([1]),ChompBoard([2,2])]
invariants = [ChompBoard.number_of_rows, ChompBoard.number_of_columns,
  ChompBoard.number_of_cookies]
invariant_of_interest = invariants.index(ChompBoard.number_of_cookies)
conjecture(objects, invariants, invariant_of_interest)
```

The program was given as input (or *knows*) the two positions `[1]` and `[2,2]`. It is using three invariants: `number_of_rows`, `number_of_columns`, and `number_of_cookies`. The chosen invariant to generate conjectured bounds for is `number_of_cookies`. The procedure `conjecture` calls the program; the default is to conjecture upper bounds for the chosen invariant; to generate lower bound conjectures, the user would add the parameter `upperBound = False` to this function call. (The operator list is currently hard-coded: a subset can easily be input, but addition to this list is not a simple input option). The program then produces the following two conjectures:

```
number_of_cookies(x) <= 2*number_of_rows(x)
number_of_cookies(x) <= number_of_rows(x)^2
```

(The caret symbol "`^`" is the program's symbol for exponentiation.) These conjectures may naturally be interpreted as being true for all objects of the input type: arbitrary Chomp positions. It happens to be the case here that these two positions are square boards—so the user could also interpret the conjectures as being conjectures for square boards. But the conjectures must necessarily be given *some* interpretation.

Note that both produced conjectures are true for both the input Chomp positions `[1]` and `[2,2]`. While the right-hand-side of the first conjecture gives the exact value for the number of cookies for position `[2,2]`, it does not give the exact value for position `[1]`. So it is possible that another conjecture could give an improved value or bound. In fact the second conjecture gives the exact number of cookies for position `[1]`. That is, the second conjecture is *significant* with respect to the previously produced conjectures and the input objects.

The general approach to generating conjectures is as follows.

1. *Produce a stream of inequalities with evaluable functions of the invariants on each side of the inequality symbol.* Some of these will pass the truth and significance tests and be stored as conjectures.

2. *Initialize an initial collection of game positions.* These can be as few as one.

3. *Generate conjectures that are true for all stored game positions and significant with respect to these game positions and the previously stored conjectures.* Pass each generated statement to the truth and significance tests. The program needs a stopping condition. The best case is that, for each game position, there is at least one conjecture that gives the exact value for that position. In this case there is no possibility of improving the current conjectures—in the sense that no other conjectures can make

better predictions about the values of the existing game positions—exact predictive power for all game positions has been achieved. In the case where this natural stopping condition is never attained, another stopping condition will be required. One possibility is to simply stop making conjectures after some hardcoded or user-specified time.

4. *Remove insignificant conjectures.* After a conjecture is added to the store of conjectures, it may be the case that another conjecture in the store is no longer significant. If conjectured upper bounds (for example) for an invariant α are being generated then a conjectured bound α_i in the conjectures store is significant, with respect to the stored game positions, if and only if there is a game position B such that $\alpha_i(B) < \min\{\alpha_j(B) : j \neq i\}$, that is, if and only if, there is an object B where the bound gives a better predicted value for $\alpha(B)$ than any other conjectured bound does. Insignificant conjectures are then removed.

If no-longer-significant conjectures are removed whenever significant conjectures are added to the conjectures store, the number of conjectures (of any particular form) cannot exceed the number of example game positions. This puts an important limit on the number of produced conjectures.

4. Chomp Conjectures & Move Heuristics

Every Chomp position either has a winning strategy for the *n*ext player to play—an N-position—or for the player that did *p*reviously play—a P-position (the language of N- and P-positions comes from the standard book on combinatorial games (Berlekamp, Conway, & Guy, 2004)). We are primarily interested in developing P-theory: that is, in the theory and identification of P-positions. The reason is that, when a player makes a move, assuming that she has a winning strategy, she must choose one of the remaining cookies to remove. Each choice will yield a board position for her opponent—these positions are the *reachable* positions from the current position. If she plays perfectly she will give her opponent a P-position. So her perfect play boils down to identifying the P-positions that are reachable from her current position.

A preliminary conjecture-based Chomp-player idea was proposed by Larson and Van Cleemput (2016). The idea there was to conjecture a theory of N-positions and to use this theory to avoid giving one's opponent an N-position. This idea may still have some value. And might be used in conjunction with the current approach.

The simplest P-position `[1]` consists of a board whose only cookie is the poison cookie. Any other single-row Chomp position is necessarily an N-position. *L-boards* are another example of a simple—and analyzed—Chomp board position; these are boards whose only cookies are in the first row and first column. The board `[2,1]` is an L-board with sides of length two; it is a P-position. The board `[3,1]` is an L-board with sides both of length two and three; it is an N-position. It can be argued that all L-boards with equal length sides are P-positions, while all L-boards with unequal length sides are N-positions. The strategy for a player facing an L-board with unequal length sides is to eat a cookie so that the resulting position is an L-board with equal length sides. This process can be iterated:

each intermediate board is an L-board, and the initial player can, on each turn, pass her opponent an L-board with equal length sides.

Trivially, a position is a P-position if and only if it is not an N-position. More usefully, a position is an N-position if *there is* a reachable position that is a P-position. And a position is a P-position if *every* reachable position is an N-position. This idea can be used recursively to determine if any given position is a P-position. And we used this idea to create a small database of both N-positions and P-positions that we used in testing generated conjectures. This recursive position-evaluation idea is not generally useful for humans—and if the position is sufficiently large no program will be able to carry out this computation in a reasonable amount of time either.

Generated conjectures for Chomp P-positions might at least approach an ideal and effectively computable Chomp P-theory. We initially determined a small number of known P-positions, including boards `[2,1]`, `[1]`, `[5,1,1,1,1]`, `[3,2]`, and `[7,6]`. These were the input examples for the earliest conjectures. The process was iterated whenever we found a counterexample to any of the conjectures. That is, as we found counterexamples to generated conjectures, those counterexamples were added to the list of Chomp P-position Conjecturing program input. Conjecturing was never given more than 60 examples total to use in any run of conjecture generation. These examples certainly constitute *knowledge*—but it is not the expert knowledge of an expert Chomp-player (none of us attained any Chomp skill at all during this project). There are a few reasons why we keep the number of input examples relatively small—and limited to ones that are in some sense *important*; this is ordinary practice in mathematics (we are mathematicians).

We then conjectured upper and lower bounds for each of these defined invariants in terms of the other invariants. Produced conjectures are true for all stored P-positions in our database. These can then be divided into two classes: conjectures that are false for at least one (recursively computed and) stored N-position, and conjectures that are true for all stored N-positions. In the later case, the conjecture is necessarily true for *every* stored position—and is thus uninformative. These were then eliminated; the remaining conjectures are, thus, genuine P-theory conjectures.

The following conjectures come from a list produced for discussion in a summer research project. They were selected for publication only for the reason that they could be printed comfortably and readably below. Internally the conjectures are all represented as labeled trees. We define the complexity of these expression-trees as the number of nodes in its representation. As mentioned earlier, an invariant itself has complexity 1, an expression formed by a unary operator applied to a single invariant has complexity 2, etc. The complexity of the following conjectured bounds are no more than 7.

The first conjecture below is proved—and constitutes new Chomp *knowledge*. The proof of the second is symmetric. The rest are open problems.

```
14. number_of_cookies(x) >= 2*number_of_columns(x) - 1
15. number_of_cookies(x) >= 2*number_of_rows(x) - 1
24. number_of_cookies(x) <= 2*number_of_columns(x) + 2*squareness(x)
29. full_rectangle(x) <= row_product(x)^2
33. rows_of_different_length(x) >= log(number_of_columns(x))/log(10)
36. rows_of_different_length(x) <= average_cookies_per_row(x) + 1
```

```
38. squareness(x) >= cookies_inside_ratio(x)
47. row_product(x) >= number_of_rows(x)
48. row_product(x) >= smallest_row_size(x)^number_of_rows(x)/
    rank_ratio(x)
55. row_product(x) <= number_of_rows(x)^number_of_cookies(x)
64. column_product(x) >= (squareness(x) + 1)*number_of_rows(x)
87. average_cookies_per_column(x) <=
    2*squareness(x)/number_of_columns(x) + 2
90. average_cookies_per_row(x) >= -1/2*duplicate_rows(x) +
    1/2*number_of_columns(x)
```

Consider a 9×9 initial board. This is a full rectangle and is, by Gale's theorem, an N-position. In fact, one winning move is to take the cookie in the second row and second column. This yields the L-board `[9,1,1,1,1,1,1,1,1]`. Now the program will consider the 81 possible moves consisting of taking a cookie from any of the 81 spots that have cookies on them, and then removing cookies below and to the right. One of these positions will be the position `[9,1,1,1,1,1,1,1,1]`. Table 1 contains values for the invariants for this position. It is now easy to check that all the listed conjectures, for instance, are true for

Invariant	Value	Invariant	Value
number_of_cookies	17	rows_of_different_length	2
number_of_columns	9	average_cookies_per_row	$\frac{17}{9}$
number_of_rows	9	cookies_inside_ratio	0
squareness	0	row_product	9
full_rectangle	81	average_cookies_per_column	$\frac{17}{9}$
row_product	9	duplicate_rows	7
rank_ratio	$\frac{2}{9}$		

Table 1: Values of invariants for position `[9,1,1,1,1,1,1,1,1]`.

the position `[9,1,1,1,1,1,1,1,1]`. The move initiated by taking the cookie in the second row and second column is satisfied by the conjectured theory.

A number of conjectures that were trivially true or very easy to prove (for instance that, for any P-position, `number_of_cookies` >= min (`number_of_rows^2`, `number_of_columns` + 1)). Conjectures 14 and 15 above turned out to be non-trivial—and true. We prove Conjecture 14 below. A proof of Conjecture 15 follows similarly. For a board position B let $n = n(B)$ be the `number_of_cookies` on the board, and let $c = c(B)$ be the `number_of_columns` on the board. Notice that the theorem gives a necessary condition for P-positions and thus adds to existing P-theory. Note too that the condition in the theorem does not hold for arbitrary N-positions: in particular it is false for the position `[3]` (here $n = 3$, and $c = 3$) and false for almost every other one row position. The theorem shows that the conjectures may be generally true—not only true for the input examples but true for *every* P-position.

Theorem 4.1. *For any P-position B, $n(B) \geq 2c(B) - 1$.*

Proof. The statement is clearly true for small board positions (for instance the P-position `[1]`). Suppose then that $n(B) \geq 2c(B) - 1$ for all P positions B with less than k cookies. Let B be a P-position with k cookies. If every column contains two or more cookies we are done (as $n(B) \geq 2c(B)$ in this case). So we can assume there is at least one column with a single cookie. In particular the right-most column must have a single cookie. Any move from a P-position must obtain an N-position. So let B' be the N-position obtained by the first player to move by taking that right-most cookie in the first row. So $n(B') = n(B) - 1$ and $c(B') = c(B) - 1$.

The main idea in the proof is *strategy-stealing*. There must now be a move that obtains a P-position B''. That move *cannot* consist of taking a cookie from the first row—otherwise it could have been obtained directly from position B—but only N-positions are reachable from B. So the move to obtain P-position B'' from N-position B' must begin by removing a cookie from some row below the first row. Assume t cookies were removed. So $n(B'') = n(B') - t$ and in turn we have $n(B) = n(B'') + t + 1$.

Importantly the number of columns of B' and B'' must be the same; so $c(B) = c(B') + 1 = c(B'') + 1$. And since B'' is a P-position with fewer than k cookies it follows from the inductive hypothesis that $n(B'') \geq 2c(B'') - 1$. Substituting we get $n(B) - t - 1 \geq 2(c(B) - 1) - 1$. Simplifying yields $n(B) \geq 2c(B) + t - 2$. And since $t \geq 1$ for any legal move it follows that $n(B) \geq 2c(B) - 1$, which was to be shown. □

The conjectures are interpreted to be true for every P-position (that is, they are only known to be true for the Chomp positions in the program's input list—and then generalized in some way. If the input positions are all P-positions one natural generalization is to *all* P-positions. Of course the set of input objects can be correctly described by infinitely many different properties). Ideally, if a reachable position is a P-position there will be a move consistent with all of the conjectures. The move the player should make is to take *any* cookie that yields a reachable P-position: just examine all reachable positions and check them against the conjectured theory, and choose any position that satisfies all the theory statements.

5. Discussion

We have demonstrated how a program can produce conjectures using relatively little data and that would supply human-comprehensible reasons (reasons that are relatively short statements) for a game-playing program. Chomp is our example—but these ideas apply to any sufficiently similar game. The conjectures specify relations between game board invariants that can used to choose a move. Suppose again a player is given a 9×9 full rectangle of cookies. She wants to make a move that yield a P-position. She will base her move on conjectures that capture her Chomp knowledge. Assume these are the batch of reported conjectures above. One possible move is to take the first cookie in the second row yielding the position `[9]`, but this position violates the (true, proved) conjecture `number_of_cookies(x) >= 2*number_of_columns(x)-1`. Thus she might reject that possible move for the reason that, according to her conjectural beliefs, it does not yield a P-position.

Since the underlying conjecturing program is domain independent, it may be that a wide variety of intelligent behaviors may admit similar theories produced by generating and using conjectures—and useable in making decisions. When a conjecture-based game-playing program gives a false reason, this can be addressed in the code: falsity is demonstrated by examples—and these (in this sense) important examples can be added to the code. If a conjectured reason is false, that means there must be a board position that is a counterexample. This can be added to the program, conjectures can be regenerated—and the false conjecture will disappear from the store of conjectured knowledge. The new theory will be used for justification of future moves in future games. That is, just as a human player is not born fully-formed and at maximum ability, a computer player using conjectures as reasons-for-action will evolve in interaction with humans (and other intelligent agents), and will add new significant positions to its memory. It will improve in interaction with other intelligent agents. Conjecture-based programs—and humans—learn from mistakes, remember important examples, and generate new theories that respond to newly learned examples.

There are a few noteworthy features of this approach:

1. It does not rely on processing databases of millions of games—no human has ever done this. It uses a database with a relatively small number of board positions

2. The stored board positions are not randomly selected but are typically those that have been counterexamples to previous conjectures. Leveraging counterexamples is standard practice in mathematical research—they must be responded to and often lead to improved theory (and is exemplified by the number of books with titles like *Counterexamples in Analysis*). Perhaps leveraging counterexamples is a more generally important cognitive strategy.

3. It produces conjectures (statements, reasons) for rejecting possible move choices that are arguably *explainable*. In most instances the conjectures are no more complex than theorems in mathematics texts. Of course an "explanation" here requires understanding the statement-terms, and some experience with examples—just as any other kind of technical statement. Similarly, despite the apparent simplicity of Newton's Laws, their use as explanations is not immediately apparent to physics novices.

These experiments are first steps. Better Chomp conjectures can presumably be produced with better ingredients. This might include:

1. Adding more invariants to the program; the best invariants might be ones that capture features that human players already pay attention to, or possibly strategic concepts. The invariants we used were entirely made up on our own, without any knowledge of strategic concepts.

2. Adding property-relation conjecturing ability. In another paper of this series we explained and demonstrated the design of a program that conjectures relations between the properties of an object (Larson & Van Cleemput, 2017). This might better capture position features that humans think about.

3. Using positions from published expert board games—this is part of our shared Chomp knowledge. There is no evidence that humans have any special ability to evaluate or

compute whether a board position is a P- or N-position. They must surmise them from the success of the players. A real player can, for instance, typically only assume that a position from which a known strong player won is an N-position. Of course, these judgments can be changed, and reevaluated as new knowledge is obtained. A position once believed to be an N-position can be removed from our store of known N-positions.

4. Add known Chomp theory. This will presumably lead to better move choices. Existing published Chomp P-theory is currently limited (as mentioned, essentially, to L-positions, and one- and two-rowed board positions). Chomp is a combinatorial game. So general considerations from the theory of combinatorial games could also be useful (Berlekamp et al., 2004).

Again, the underlying conjecturing program used here is domain-independent. We hope other researchers will experiment with using the approach demonstrated here (and our conjecturing program) in designing machines that generate and use conjectures that can be used as reasons or explanations.

Acknowledgements. The authors would like to thank the referees of this paper for their careful and useful comments. In particular, the proof of the theorem was greatly simplified.

References

Berlekamp, E. R., Conway, J. H., & Guy, R. K. (2004). *Winning ways for your mathematical plays*. AK Peters.

Epstein, S. (1994). For the right reasons: The FORR architecture for learning in a skill domain. *Cognitive science*, *18*(3), 479–511.

Epstein, S. (2001). Learning to play expertly: A tutorial on Hoyle. *Machines that learn to play games*, 153–178.

Fajtlowicz, S. (1995). On conjectures of Graffiti. V.. In *Graph Theory, Combinatorics, and Algorithms, Vol. 1, 2 (Kalamazoo, MI, 1992)*, Wiley-Intersci. Publ., pp. 367–376. Wiley, New York.

Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, *7*(2), 179–188.

Gale, D. (1974). A curious Nim-type game. *American Mathematical Monthly*, 876–879.

Hutchinson, L., Kamat, V., Larson, C. E., Mehta, S., Muncy, D., & Van Cleemput, N. (2018). Automated conjecturing VI: Domination number of benzenoids. *MATCH: Communications in Mathematical and in Computer Chemistry*, *80*(3), 821–834.

Langely, P., Simon, H. A., Bradshaw, G. L., & Zytkow, J. (1987). *Scientific Discovery: Computational Explorations of the Creative Process*. MIT Press, Cambridge, MA.

Larson, C. E. (2005). A survey of research in automated mathematical conjecture-making. In *Graphs and Discovery*, Vol. 69 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pp. 297–318. Amer. Math. Soc., Providence, RI.

Larson, C. E., & Van Cleemput, N. (2016). Automated conjecturing I: Fajtlowicz's Dalmatian heuristic revisited. *Artificial Intelligence*, *231*, 17–38.

Larson, C. E., & Van Cleemput, N. (2017). Automated conjecturing III: Property-relations conjectures. *Annals of Mathematics and Artificial Intelligence*, *81*(3), 315–327.

Lenat, D. B. (1977). The ubiquity of discovery. *Artificial Intelligence*, *9*(3), 257–285.

McCune, W. (1997). Solution of the Robbins problem. *Journal of Automated Reasoning*, *19*(3), 263–276.

Ratterman, M., & Epstein, S. (1995). Skilled like a person: A comparison of human and computer game playing. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, pp. 709–714.

Schmidt, M., & Lipson, H. (2009). Distilling free-form natural laws from experimental data. *Science*, *324*(5923), 81–85.

Simon, H. A., & Newell, A. (1958). Heuristic problem solving: The next advance in operations research. *Operations Research*, *6*(1), 1–10.

Todorovski, L., Džeroski, S., & Kompare, B. (1998). Modelling and prediction of phytoplankton growth with equation discovery. *Ecological Modelling*, *113*(1), 71–81.

Turing, A. (1948). A survey of research in automated mathematical conjecture-making. In Copeland, J. B. (Ed.), *The Essential Turing (2004)*, pp. 395–432. Oxford University Press.

Zeilberger, D. (2001). Three-rowed CHOMP. *Advances in Applied Mathematics*, *26*(2), 168–179.