Automated Conjecturing I: Fajtlowicz's Dalmatian Heuristic Revisited (Extended Abstract) *

Craig Eric Larson[†1], **Nico Van Cleemput**[2]
[1] Department of Mathematics and Applied Mathematics,Virginia Commonwealth University, Richmond, VA USA
[2]Department of Applied Mathematics, Computer Science and Statistics, Ghent University, Ghent, Belgium
clarson@vcu.edu, nico.vancleemput@gmail.com

Abstract

This condensed summary highlights the results of a 2016 AIJ paper reporting on a successful general-purpose conjecturing program.

1 Introduction

We have reimplemented Fajtlowicz's useful but little-known Dalmatian heuristic for the automation of mathematical conjecture-making (the first-ever reference in *Artificial Intelligence*, for instance, is our own paper [Larson and Van Cleemput, 2016]). The heuristic is general and can be used to conjectured relations between real number invariants of any objects, mathematical or otherwise. We have also implemented an idea to include existing theorems in the program; when used in this way the program is guaranteed to produce statements that are not implied by existing mathematical knowledge.

Our program often makes interesting and useful conjectures on the basis of only a few examples. Humans, ordinarily and of necessity, make decisions based on very limited data. A general automated conjecture-making module that can make plausible and useful guesses based on limited data may be a central architectural feature in the design of machines that are intelligent. Guesses can be used, for instance, to constrain a search of possible actions. Fajtlowicz introduced his Dalmatian heuristic for the automation of mathematical conjecture-making more than 20 years ago [Fajtlowicz, 1995]. Simply put, the heuristic is to produce a considered mathematical statement if it is both true—with respect to some given examples (matrices, integers, graphs, etc.)—and if the statement gives new information about those objects, in particular, if it says something about at least one of the objects which is not implied by any other stored statement or conjecture.

Our experiments in implementing and applying this heuristic, including in domains where the authors have no more knowledge than anyone who has browsed a textbook or reference book, suggest the following conclusions:

1. Successful mathematical discovery heuristics can be applicable in a variety of mathematical domains.
2. Good conjectures can be based on very limited data.
3. Mathematical discovery programs should aim to produce conjectures that address and advance pre-existing mathematical questions.
4. Intelligent conjecture-making programs for a domain do not require developer expertise in that domain.

We see conjecture-making—and conjecture-revision in the face of contradictory data (counterexamples)—as a central feature of intelligence. We make guesses, based on our previous experience in relevantly similar situations, learn that our guesses are wrong, revise them, and test them against our experience.

We provide a program for experimentation and further development. Readers are encouraged to make their own explorations. Our program CONJECTURING is available at: `nvcleemp.github.io/conjecturing/`. It functions as a package of the Sage open source mathematical software program [Stein, 2008]. Sage is intended as a free replacement for general mathematical software programs such as Maple, Matlab and Mathematica. Scripts that generate the conjectures for the runs of our program described in this paper are also available at this site.

Turing, famously, proposed the idea of designing intelligent machines as an engineering problem, and proposed a test for evaluating the success of such machines. In 1948 he suggested designing machines to do mathematical research as a starting point: mathematical research certainly requires intelligence and, it would be a good starting point as mathematical research would "require little contact with the outside world" [Turing, 2004]. In the 1950s Newell and Simon developed the Logic Theorist program that could prove (some) theorems in first-order logic, and went on to predict that a computer would discover and prove an important mathematical theorem within another decade [Simon and Newell, 1958]. Success did not come quite that quickly—but there has been

*This is an extended abstract of [Larson and Van Cleemput, 2016]

†Corresponding Author

significant progress in many areas of automating mathematical discovery, and there is no theoretical impediment to continued improvement. There is every reason to believe that Newell and Simon's prediction will be achieved—and likely sooner rather than later.

2 The Dalmatian Heuristic

Fajtlowicz's Dalmatian heuristic is used to conjecture relations between real number invariants of objects. Many common object-types, including graphs, natural numbers, and matrices, have associated real number invariants. The numerical invariants of a graph include the number of vertices of the graph, its number of edges, the maximum degree of any of the vertices, among numerous others.

It is possible to generate conjectures using only a single stored object. Counterexamples to existing conjectures can be added as additional objects. On this approach, all objects in the database are included exactly because they had some theoretical value—no object is included arbitrarily. Fajtlowicz suggests that this approach may have its own benefits when conducting research [Fajtlowicz, 1995]. The produced conjectures are based on a limited number of examples of objects of the given type.

Let $\mathcal{O}_1, \ldots, \mathcal{O}_n$ be examples of objects of a given type. Let $\alpha_1, \ldots, \alpha_k$ be real number invariants. And let α be an invariant for which conjectured upper or lower bounds are of interest. An unlimited stream of algebraic functions of the invariants can then be formed: $\alpha_1+\alpha_2$, $\sqrt{\alpha_1}$, $\alpha_1\alpha_3$, $(\alpha_2+\alpha_4)^2$, etc. (One natural way to do this, and our own approach, is to grow trees representing these expressions with operators representing algebraic operations on the non-leaf nodes—with the number of sub-nodes equal to the arity of the operator—and invariants on the leaf nodes.) These expressions can then be used to form conjectured bounds for α. If we are interested in upper bounds for α, say, we can form the inequalities $\alpha \leq \alpha_1 + \alpha_2$, $\alpha \leq \sqrt{\alpha_1}$, $\alpha \leq \alpha_1\alpha_3$, $\alpha \leq (\alpha_2 + \alpha_4)^2$, etc.

These inequalities can be interpreted as being true for all the objects of the given type. That is, the inequality $\alpha \leq \alpha_1 + \alpha_2$ can be interpreted as, "For every object $\mathcal{O}$, $\alpha(\mathcal{O}) \leq \alpha_1(\mathcal{O}) + \alpha_2(\mathcal{O})$." A conjectured upper bound u is only added to the database of conjectures if the bound passes the following two tests.

1. (*Truth test*). The candidate conjecture $\alpha \leq u$ is true for all of the stored objects $\mathcal{O}_1, \ldots, \mathcal{O}_n$, and
2. (*Significance test.*) There is an object $\mathcal{O} \in \{\mathcal{O}_1, \ldots, \mathcal{O}_n\}$ such that $u(\mathcal{O}) < \min\{u_1(\mathcal{O}), \ldots, u_r(\mathcal{O})\}$, where $u_1, \ldots, u_r$ are the currently stored conjectures. That is, the candidate conjecture would give a better bound for $\alpha(\mathcal{O})$ than any previously conjectured (upper) bound.

These criteria capture what Fajtlowicz calls the "Principle of the Strongest Conjecture": make the strongest conjecture for which no counterexample is known. By design, the truth test guarantees that the program does not know a counterexample, and the significance test guarantees that each conjectured bound is "stronger" (gives a better bounding value) than any other—at least for a single object known to the program.

3 The Program

An expression generating program (written in C for speed) is at the heart of our program. In this context an expression is just a rooted, labeled binary tree, that is, a rooted tree where each node has at most two children and each node with, respectively, two, one or no children is labeled with, respectively, a binary operator, a unary operator or an invariant. The expressions are generated according to increasing complexity, which is defined as twice the number of binary operators plus the number of unary operators. The program uses the algorithm described in [Peeters, 2008] and the numbers of generated structures have been compared to the implementation in [Peeters *et al.*, 2009]. The generated expressions are tested for being true for the provided invariant values (*truth test*) and can then be handed over to an internal heuristic or can just be output. Internally we have implemented two conjecture-making heuristics: the Dalmatian heuristic and—for testing purposes—the heuristic described in [Peeters, 2008].

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 objects.* These can be as few as one.
3. *Generate conjectures that are true for all stored objects and significant with respect to these objects 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 object, there is at least one conjecture that gives the exact value for the object. 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 objects—exact predictive power for all objects 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 objects, if and only if there is an object $\mathcal{O}$ such that $\alpha_i(\mathcal{O}) < \min\{\alpha_j(\mathcal{O}) : j \neq i\}$, that is, if and only if, there is an object $\mathcal{O}$ where the bound gives a better predicted value for $\alpha(\mathcal{O})$ than any other conjectured bound does. Insignificant conjectures are then removed.

The following steps are not necessary to the conjecturing process, but arguably lead to better conjectures.

1. *Search for a counterexample to any of these conjectures.* This can be done by a human or automated in some way. In the case of number theory conjectures, the conjectures

can easily be checked by testing the conjectures for each integer from 1 up to an arbitrary large integer. In the case of other objects, it will usually require some work to generate a stream of distinct objects. In the case of graphs, McKay's *geng* provides a stream of examples from graphs with a single vertex up to any user-specified number of vertices.

2. *Add the counterexample, and repeat the conjecture generating and testing process*. The program can never make more conjectures than the number of objects it has stored: the reason is exactly because each conjecture in the conjectures store must give a better bound for at least one stored object than any other conjecture does.

   If there were, for instance, two stored objects and three conjectured bounds, at least one of the conjectured bounds could not possibly be significant: at best one of the conjectures could be the best bound for one of the objects and another for the second object—but the third conjecture would have no possible remaining objects for which it could give the unique best predicted value; this conjecture would have been removed as *insignificant*.

From the point of view of a user of our program, the required *inputs* of the program are three:

1. A list of objects. The type is arbitrary, but they will usually all be of the same type. To get meaningful results they will all represent the same mathematical type of object. For instance, if you want to generate conjectures about graphs, and `c5`, `k5` and `petersen` are pre-defined graph objects, you would define `objects` = [`c5`, `k5`, `petersen`], and give `objects` as a parameter to the program.
2. A list of invariants. These must be functions that are defined for the type of objects in the `objects` list. For instance, if `radius`, `size` and `order` are predefined real-valued graph functions, you would define `invariants` = [`radius`, `size`, `order`] and give `invariants` as a parameter to the program.
3. A positive integer listing the position of the invariant in the list of `invariants` that you would like to conjecture bounds for from the list of `invariants`. For instance if conjectures for the `radius` of a graph, the user would enter 0 in the list of parameters to the C program.

4 Bounds for the Graph Theoretic Domination Number

This is an example of one of our experiments. A *dominating set* in a graph is a set D such that every vertex of the graph which is not in D is adjacent to at least one vertex in D; the *domination number* of a graph is the cardinality of a minimum dominating set [Haynes *et al.*, 1998]. Computing the domination number of a graph is intractable (NP-hard) and currently impossible for general graphs of even moderate size. Conjectured bounds for the domination number are of theoretical interest—bounds which are functions of efficiently computable invariants are also of practical interest—these can lead to speed up of domination number computations.

The objects are connected graphs. The invariants we started with included `domination_number`, `matching_number`, `annihilation_number`, `girth`, `radius`, `fractional_independence_number`, `average_distance`, `diameter`, `order`, `size`, `szeged_index`, `wiener_index`, `average_degree`, `min_degree`, `max_degree`, and `residue`. Many of these are standard graph theoretic invariants that can be found in introductory texts such as [West, 2001]. These invariants were either built-in Sage functions or were coded by us as Sage procedures. For acyclic graphs, girth was set to infinity. The Szeged and Wiener indices are of special interest in chemical graph theory. The *fractional independence number* is the optimum solution to the relaxation of the independence number linear program (and thus an upper bound for the independence number). The *annihilation number* is a degree sequence upper bound for the independence number introduced by Pepper [Pepper, 2009; Larson and Pepper, 2011], and the *residue* is a degree sequence lower bound for the independence number introduced by Fajtlowicz [Favaron *et al.*, 1991].

The three invariants listed after `domination_number` are known upper bounds for the domination number and were eventually removed in order to try to generate better upper bound conjectures. Manual removal of invariants in this way is no longer required: the inclusion and use of known bounds would have precluded the initial production of these conjectures.

We used McKay's program *geng* [McKay, 2007] to generate all graphs up to some (small) specified order in a loop to automatedly find counterexamples to generated conjectures and, thus, automatedly improve the produced conjectures. In our run generating upper bound conjectures for the domination number, the program ended up with four examples (found by this automated search for counterexamples) and the conjecture that the domination number of a graph is no more than its matching number. The conjecture exactly predicted the true value of the domination number of these four graphs—and, hence, the program stopped. This is a known (and not difficult to prove) fact about the domination number. Again, if existing theory had been included, this conjecture could not have been made.

In the next run, we removed `matching_number` from the list of invariants and the program generated the three conjectures in Table 1. The first two we knew to be true. The third is false: Ryan Pepper found a 24 vertex counterexample.

After adding Pepper's counterexample, we generated another run of upper bound domination conjectures. These are in Table 2. Stephen Hedetniemi, a co-author of the standard reference on domination [Haynes *et al.*, 1998], points out that the second of these conjectures is false for K_1 and K_2—we

1.	`domination_number(x)`	$\leq$	`fractional_independence_number(x)`
2.	`domination_number(x)`	$\leq$	`annihilation_number(x)`
3.	`domination_number(x)`	$\leq$	`residue(x)` + 1

Table 1: Upper bound conjectures for the domination number of a graph.

1.	domination_number(x)	$\leq$	1/2*order(x)
2.	domination_number(x)	$\leq$	size(x) - 1
3.	domination_number(x)	$\leq$	2*diameter(x) - 1
4.	domination_number(x)	$\leq$	diameter(x)^2
5.	domination_number(x)	$\leq$	girth(x)^2
6.	domination_number(x)	$\leq$	residue(x) + 2
7.	domination_number(x)	$\leq$	-average_degree(x) + order(x)
8.	domination_number(x)	$\leq$	max(radius(x), average_distance(x)^girth(x))
9.	domination_number(x)	$\leq$	2*residue(x) - 1
10.	domination_number(x)	$\leq$	min_degree(x) + residue(x)
11	domination_number(x)	$\leq$	2*diameter(x) - radius(x) + 2
12.	domination_number(x)	$\leq$	(min_degree(x) + 1)^(residue(x) - 1)
13.	domination_number(x)	$\leq$	max(residue(x), -girth(x) + order(x))
14.	domination_number(x)	$\leq$	max(diameter(x), order(x) - 2*radius(x))

Table 2: Upper bound conjectures for the domination number of a graph.

had only been including graphs of order $n \geq 3$ in our automated counterexample search—and trivially true for graphs or order $n \geq 3$. The truth of Conjecture 7 follows from a well-known result. He also provided counterexamples to Conjectures 3, 4, 5, and 11.

5 Discussion & Future Work

Here are two observations from our use of the program.

Successful conjecture-making programs do not require domain-specific heuristics. The description of the Dalmatian heuristic does not refer to any particular branch of mathematics, or even to mathematical object-types. We have demonstrated its general utility in graph theory, number theory and matrix theory, and in characterizing game positions. The authors specifically generated conjectures for mathematical areas in which we had no expert knowledge. We paged through relevant books looking for invariants and to try to determine invariants for which experts would be interested in conjectured bounds.

Knowledge of existing theorems can improve the conjectures produced by a conjecture-making program. This is knowledge that experts would have—but not "expert knowledge"—anyone can page through the relevant texts and papers to find these theorems. Knowledge of all examples of objects that have appeared in the literature of a domain would also improve the conjectures. For the program described here, it would guarantee the truth of any produced conjecture with respect to at least these objects. It would also be useful to have an an "intelligent" counterexample-finder. We do not know of one—or whether these would require domain-specific heuristics. The object generators used in our research all have the same underlying idea. These are finite structures and they can be systematically generated for all objects of a desired "size". No expert knowledge is required here. Generators like *geng* – used here for generating graphs – are simply more efficient than ones non-experts can write. A graph of order n, for instance, is simply a symmetric 0-1 matrix. A non-expert can easily write a program that generates all symmetric 0-1 matrices up to any order.

There is some sense in which domain-specific knowledge can be of use in improving conjecture-making programs: experts do not need to consult the literature to find invariants and examples, and they can write more efficient object-generators. Nevertheless we know of no example of a successful conjecture-making program that uses domain-specific heuristics. And we only claim here that domain-specific heuristics are not necessary.

Success of conjecturing programs is by design. Scientific discovery, in general, is the result of effort directed at specific questions of interest; we are not aware of any case of discovery which cannot to traced back to work on specific problems.

Development of a program that contributes to scientific discoveries requires knowing what counts as a contribution to scientific discovery; a successful discovery program must make such a contribution. Scientists and mathematicians must address this issue in their own work: to make a scientific discovery you must first know what the open questions are and which ones are the most central. And not all scientific and mathematical research is of equal value.

The only way to determine the value of mathematical research is to engage the community of mathematical researchers and users of mathematics about how the research is connected to existing mathematical questions and what potential consequences of the research are; there is no external criteria for judging the value of mathematical research. Many mathematical papers explicitly address an existing mathematical problem—they intend to make a contribution either by answering an outstanding question, by helping to better understand the problem or its difficulties, or by developing tools that might be used in attacking the problem.

Our next goal is to advance research on a specific question of continuing mathematical research: bounds for the *independence* number of a graph. We plan to add all known independence number bounds, and produce conjectures that are, in turn, better than existing theory, for at least one graph. A proof of any of these would be a proof that automated conjecturing programs can make genuine, scientifically interesting, contributions.

Since the publication of [Larson and Van Cleemput, 2016] we have produced a property-relations conjecturing program, which works analogously to the described invariant-relations conjecturing program, conjectured and proved a new theorem for P-positions in the combinatorial game Chomp, and discovered and proved several theorems for the domination number of benzenoid graphs (graphs representing the carbon structures of benzenoid molecules). We have also used the program to produce a "conjectured proof" of the well-known Friendship Theorem about graphs where every pair of vertices has exactly one common neighbor [Aigner and Ziegler, 2010]. This experiment suggests that it may be possible to use published knowledge together with conjectures to produce *semantic proofs* (as opposed to the *syntactic proofs* of traditional automated theorem-proving programs) of mathematical conjectures. We do not yet know the full possibilities—or limitations—of these ideas.

Our dream is to code all properties, invariants, graphs, and theorems that have appeared in the graph theory literature. This would be a long-term, massive project, which will ultimately require the work of a community of interested researchers. But then, in a well-defined sense, no human could produce a simpler conjecture than our program could—that is true for all published graphs, and which improves on all known theorems. A well-funded experiment along these lines could lead the way to a new era of mathematical research.

References

[Aigner and Ziegler, 2010] Martin Aigner and Günter M. Ziegler. *Proofs from the Book, 4^{th} ed.*, volume 274. Springer, 2010.

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

[Favaron *et al.*, 1991] Odile Favaron, Maryvonne Mahéo, and Jean-François Saclé. On the residue of a graph. *J. Graph Theory*, 15(1):39–64, 1991.

[Haynes *et al.*, 1998] Teresa W. Haynes, Stephen T. Hedetniemi, and Peter J. Slater. *Fundamentals of Domination in Graphs*, volume 208 of *Monographs and Textbooks in Pure and Applied Mathematics*. Marcel Dekker Inc., New York, 1998.

[Larson and Pepper, 2011] Craig E. Larson and Ryan Pepper. Graphs with equal independence and annihilation numbers. *Electronic Journal of Combinatorics*, 18(1), 2011.

[Larson and Van Cleemput, 2016] Craig E. Larson and Nico Van Cleemput. Automated conjecturing I: Fajtlowicz's Dalmatian heuristic revisited. *Artificial Intelligence*, 231:17–38, 2016.

[McKay, 2007] Brendan D. McKay. Nauty users guide (version 2.4). *Computer Science Dept., Australian National University*, 2007.

[Peeters *et al.*, 2009] Adriaan Peeters, Kris Coolsaet, Gunnar Brinkmann, Nico Van Cleemput, and Veerle Fack. GrInvIn in a nutshell. *J. Math. Chem.*, 45(2):471–477, 2009.

[Peeters, 2008] Adriaan Peeters. *GrInvIn – A Software Framework for Education and Research in Graph Theory*. 2008. Thesis (Ph.D.)–Ghent University.

[Pepper, 2009] Ryan Pepper. On the annihilation number of a graph. *Recent Advances in Applied Mathematics and Computational And Information Sciences*, 1:217–220, 2009.

[Simon and Newell, 1958] Herbert A. Simon and Allen Newell. Heuristic problem solving: The next advance in operations research. *Operations Research*, 6(1):1–10, 1958.

[Stein, 2008] William Stein. *Sage: Open Source Mathematical Software (Version 2.10.2)*. The Sage Group, 2008. `http://www.sagemath.org`.

[Turing, 2004] Alan Turing. Intelligent machinery. *The Essential Turing*, pages 395–432, 2004.

[West, 2001] Douglas B. West. *Introduction to Graph Theory, 2^{nd} edition*. Prentice Hall, 2001.