Automated conjecturing I: Fajtlowicz's Dalmatian heuristic revisited

C.E. Larson [a],[*],[1], N. Van Cleemput [b],[c]

[a] *Department of Mathematics and Applied Mathematics, Virginia Commonwealth University, Richmond, VA 23284, United States*
[b] *Department of Mathematics, European Centre of Excellence NTIS (New Technologies for the Information Society), University of West Bohemia, Univerzitní 8, 306 14 Plzeň, Czech Republic*
[c] *Department of Applied Mathematics, Computer Science and Statistics, Ghent University, Krijgslaan 281 – S9 – WE02, 9000 Ghent, Belgium*

ARTICLE INFO

ABSTRACT

We discuss a new implementation of, and new experiments with, Fajtlowicz's Dalmatian conjecture-making heuristic. Our program makes conjectures about relations of real number invariants of mathematical objects. Conjectures in matrix theory, number theory, and graph theory are reported, together with an experiment in using conjectures to automate game play. The program can be used in a way that, by design, advances mathematical research. These experiments suggest that automated conjecture-making can be a useful ability in the design of machines that can perform a variety of tasks that require intelligence.

1. Introduction

We have reimplemented Fajtlowicz's useful but little-known Dalmatian heuristic for the automation of mathematical conjecture-making (this heuristic, for instance, has never been referenced in the pages of *this journal*). The heuristic is general and can be used to conjecture relations between real number invariants of any objects, mathematical or otherwise. We include examples of conjectures in number theory, matrix theory, graph theory and the characterization of game positions. One of the number theory conjectures implies, and is stronger than, Goldbach's Conjecture. Some of the number theory conjectures seem to imply the Riemann Hypothesis. And some of the graph theory conjectures would advance the lower bound theory of the independence number of a graph, a widely-studied NP-hard graph invariant. We have also implemented an idea, suggested to us by Barry Mazur, 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 [1]. 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.

It was very successful—both in limiting the number of conjectures produced by earlier versions of his GRAFFITI program and in producing conjectures of interest to research mathematicians. His student DeLaVina reimplemented the heuristic in a program that produces conjectures that have led to research and publications by mathematicians [2]; otherwise the heuristic has not been used. Fajtlowicz made some experiments to demonstrate the domain independence of the Dalmatian heuristic; nevertheless, the predominant and best-known uses of the heuristic—in the programs of Fajtlowicz and DeLaVina—has been in the production of graph theory conjectures. But the heuristic is not specific to the production of graph theory conjectures.

Our program is open-source, written in Python and C, and implemented as a Sage package. Details about the acquisition and use of our program, the Sage open-source mathematical computing environment, and how to reproduce our results are relegated to Appendix A.

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, lead us to make several conclusions, which we will elaborate and discuss.

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.

Some of these conclusions should be surprising and, we hope, inspire new research in automated scientific discovery.

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.

2. Background & related work

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" [3]. 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 [4]. Success did not come quite that quickly—but there has been 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.

The automation of theorem proving is by far the largest and best-developed area of automated mathematical discovery research. A highlight in this area was the 1996 computer proof of the Robbins Conjecture [5]. More recently Timothy Gowers, a Fields Medalist, and likely the most accomplished mathematician to do research in automated mathematical discovery has, together with Mohan Ganesalingam, developed a theorem-proving program.[2]

Research on automated conjecture-making was initiated by Wang in the late-1950s [6]. His Program II produced thousands of statements in propositional logic that could be considered as conjectures or potential theorems. His program included heuristics for deciding which statements to output. Evaluated as a tool for advancing mathematical research, Wang's program was a failure. He wrote:

> It was at first thought that these crude principles are sufficient to cut down the number of theorems to a degree that only a reasonably small number of theorems remain. It turns out that there are still too many theorems. The number of theorems printed out after running the machine for a few hours is so formidable that the writer has not even attempted to analyze the mass of data obtained [6].

What Wang really wanted was for his program to produce a limited number of statements of interest to logicians. Wang selected a few statements to include in publication—but what was really needed was a way for the program itself to identify the interesting, useful or important statements.

The first program to make conjectures leading to published mathematical research was Fajtlowicz's GRAFFITI program [7–10,1]. An early version of GRAFFITI was called the "Sorcerer's Apprentice" [11] because the program, like Wang's, produced a large number of statements. In the Goethe poem (and the Disney *Fantasia* version with Mickey Mouse) a sorcerer's apprentice intends to use his master's spells to animate a broom to help him carry a bucket of water but he ends up with

---

[2] A preprint of their paper is available at: arxiv.org/abs/1309.4501.

so many brooms and buckets of water that the "help" is no help at all—the flood of water is a bigger mess than he had to clean up in the first place. The Sorcerer's Apprentice Problem is how to reduce the flood of potential conjectures to a useable or scannable number—how to design a program to produce just the most "significant", "interesting" or useful statements? It is not difficult to program a computer to produce an endless stream of mathematical statements. And given a stream of mathematical statements, there is a chance that some of these statements will be of mathematical interest. The problem is to produce just these ones.

This problem was remedied by Fajtlowicz's invention of his Dalmatian heuristic, implemented in early-1990s versions of GRAFFITI [12]. The Dalmatian heuristic, by design, limits both the quantity of output statements and guarantees the quality of the output statements. The program cannot make any more conjectures than the number of objects being considered (stored in the program)—so the number of generated conjectures is fundamentally limited. Each conjecture must be significant with respect to at least one object—each must provide better information about a stored object than *any* of the other conjectures. When applied to the problem of finding bounds of invariants—in cases where bounds are of pre-existing research interest—and provided with examples (objects) where existing theory does not suffice to predict the value of the invariant for the example, a program implementing the Dalmatian heuristic will produce a conjecture. In the sense that the truth of the conjecture would advance existing theory, the conjecture can be said to be interesting or significant. The utility of the output can be further improved by including existing knowledge in the program. We will discuss one experiment implementing this new idea.

The GRAFFITI program and some of its conjectures are described in Fajtlowicz's papers and in [13]. The Dalmatian heuristic first appears in [1]. DeLaVina's GRAFFITI.PC program, a successor to GRAFFITI which implements this heuristic, is described in [2]. Selected conjectures of GRAFFITI with commentary were collected by Fajtlowicz and included in his evolving report *Written on the Wall* (WoW); these were distributed by email to interested researchers.[3] The conjectures of GRAFFITI.PC are collected by DeLaVina in *Written on the Wall II* (WoW2).[4]

It is worth noting some differences between these programs. Fajtlowicz experimented with a variety of heuristics and only used a small number of objects (predominantly graphs). DeLaVina's GRAFFITI.PC could compute, maintain and use data from millions of graphs. GRAFFITI was written in Pascal, had a command-line interface, and was not generally distributed. GRAFFITI.PC was written in C++ and Visual Basic, was designed with other users in mind, has an attractive GUI interface, and was distributed to interested students and researchers. The code for these programs was not distributed. Our program shares only its use of the Dalmatian heuristic with GRAFFITI and GRAFFITI.PC. Our code is freely available to be downloaded, experimented with, modified, and used. It is a goal of this project to encourage the general use and experimentation with conjecture-making programs, and to make this easy. The use of existing bounds to improve conjecture quality is new to the described program. Fajtlowicz reports that the conjectures in [1] were based on a database of some 600 graphs; the memory available on the computers of its day were a natural limitation on the number of objects that could be used by GRAFFITI. DeLaVina often uses a database of all connected graphs up to a small number of vertices, with typically more than a million graphs. While the computers our program runs on have relatively huge amounts of memory, and our program could use large numbers of objects, we have in practice only used very small number of objects in generating conjectures. The number it can use is limited only by the memory limits of the machine it is on. We can, and have, generated millions of objects in our searches for counterexamples to its conjectures.

A variety of programs have been developed that either attempt to simulate how research mathematicians make conjectures, or that try to produce conjectures of interest to mathematicians, or both. These include the programs of Fajtlowicz and DeLaVina, as well as Lenat's AM [14–17], Epstein's GT [18,19], Colton's HR [20–23], Hansen and Caporossi's AGX [24–26], and Mélot's GRAPHEDRON [27,28]. GRAFFITI, GRAFFITI.PC and AGX have led to an especially large number of publications by mathematical researchers. There is also related and interesting work on the automation of mathematical discovery by many others including the GRAPH program of Cvetkovíc and a large group of University of Belgrade collaborators [29], Brigham and Dutton's INGRID program [30,31], the geometry programs of Bagai and collaborators [32,33], the hypergeometric series work of Wilf and Zeilberger [34], and applications of automatic recognition of integer-relations of Borwein, Bailey and their collaborators [35].

3. 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. (Some mathematical objects, including arbitrary topological spaces, do not obviously 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. The numerical invariants of a matrix would include the determinant of the matrix, its rank, the number of rows, etc. The numerical invariants of a natural number would include the number itself, its number of factors, the number of primes no more than the number, and the number of distinct primes in its unique factorization.

---

[3] A version from 2004—possibly the last version Fajtlowicz distributed—can be found on the first author's blog at: independencenumberproject. wordpress.com.

[4] Available at: cms.uhd.edu/faculty/delavinae/research/wowII/index.htm. DeLaVina also maintains a lists of papers inspired by conjectures of both GRAFFITI and GRAFFITI.PC (at: cms.uhd.edu/faculty/delavinae/research/wowRef.htm).

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 [1]. 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.

What follows is pseudocode for this MAIN LOOP of our program for the case where conjectured upper bounds for some `invariant` is desired. `upper_bound`, a potential conjectured upper bound for `invariant`, is an expression constructed from the invariants and operators included in the program. `objects` is the list of examples known to the program. The value of `invariant` and `upper_bound` applied to each $\mathcal{O}$ in `objects` is a real number; similarly, the value of `u` applied to $\mathcal{O}$, for each `u` in `conjectured_upper_bounds` and each $\mathcal{O}$ in `objects` is a real number. In the case where both the truth and significance tests are passed, `upper_bound` is added to the conjectures store `conjectured_upper_bounds`. The MAIN LOOP for the case of lower bound conjectures would parallel this.

```
def truth(invariant, upper_bound, objects):
    for O in objects:
        if invariant(O) > upper_bound(O):
            return Fail
    else:
        return Pass


def significance(upper_bound, conjectured_upper_bounds, objects):
    for O in objects:
        if upper_bound(O) < min([u(O) for u in conjectured_upper_bounds]):
            return Pass
    else:
        return Fail


def dalmatian(invariant, upper_bound,  conjectured_upper_bounds, objects):
    if truth(invariant, upper_bound, objects) == Pass and
        significance(upper_bound, conjectured_upper_bounds, objects) == Pass:
        conjectured_upper_bounds += [upper_bound]


def main(objects, invariants, invariant):
    conjectured_upper_bounds = []
    while(stopping_condition == False):
        upper_bound = generate_next_potential_upper_bound()
        dalmatian(invariant, upper_bound,  conjectured_upper_bounds, objects)
```

Here is a concrete illustration of how the Dalmatian heuristic works, with an example from number theory—as the invariants here should be generally known. We will step through the generation of conjectured lower bounds for $\pi(x)$, the number of primes no more than x. Note, here, that $\pi(5) = 3$ and $\pi(16) = 6$. So, ideally, we will generate lower bounds for $\pi(x)$ where the maximum of the conjectured lower bounds applied to the object 5 is 3, while the maximum for 16 is 6.

Table 1
Dalmatian heuristic example.

	Lower bound expression	Values	Truth test	Significance test	Conjectures store
1.	`sum_of_digits`	$5 \leq 3$ $7 \leq 6$	Fail	N/A	∅
2.	`number_prime_factors`	$1 \leq 3$ $1 \leq 6$	Pass	Pass	`number_prime_factors`
3.	`number_of_divisors`	$2 \leq 3$ $5 \leq 6$	Pass	Pass	`number_of_divisors`
4.	`sum_of_digits^2`	$25 \leq 3$ $49 \leq 6$	Fail	N/A	`number_of_divisors`
5.	`number_prime_fac-tors^2`	$1 \leq 3$ $1 \leq 6$	Pass	Fail	`number_of_divisors`
6.	`number_of_divisors^2`	$4 \leq 3$ $25 \leq 6$	Fail	N/A	`number_of_divisors`
7.	`sum_of_digits +` `number_prime_factors`	$6 \leq 3$ $8 \leq 6$	Fail	N/A	`number_of_divisors`
8.	`sum_of_digits +` `number_of_divisors`	$7 \leq 3$ $12 \leq 6$	Fail	N/A	`number_of_divisors`
9.	`number_prime_factors +` `number_of_divisors`	$3 \leq 3$ $6 \leq 6$	Pass	Pass	`number_prime_factors +` `number_of_divisors`

The initial `objects` are 5 and 16, the invariants used to form potential lower bounds are `sum_of_digits`, `number_prime_factors`, and `number_of_divisors`, while the operators are `^2`, `+`, and `+1`. Here we also assume that, after a significant conjecture is added to the conjectures store, that insignificant conjectures are removed. The order that potential lower bounds are considered corresponds to the iteration number in Table 1. First all of the invariants themselves (each represented by a tree with a root and no other nodes) are considered, followed by the application of the unary squaring operator to these invariants, followed by application of the binary sum operator applied to each distinct pair of invariants. Expressions formed with the unary "add 1" operator are never formed as the program stops after the ninth iteration, at which point the conjectures in the conjecture store exactly predict the value of $\pi(x)$ for all objects x in $\{5, 16\}$.

The considered conjecture in Iteration 1 in Table 1 is "the sum of the digits of an integer x is no more than $\pi(x)$". This is not true of all objects known to the program, does not pass the truth test, and the conjecture store remains empty. The next considered conjecture is "the number of prime factors of any integer x is no more than $\pi(x)$". This conjecture passes the truth test, as it is true for all known objects, and trivially passes the significance test. The conjectures store is updated to include this lower bound. The considered conjecture in Iteration 3 is "the number of divisors of an integer x is no more than $\pi(x)$". This is true for all of the objects and is significant as this lower bound is better than the `number_prime_factors` lower bound for at least one of the known objects; so `number_prime_factors` is added to the conjectures store. In fact it is at least as good for *all* the known objects. Since there is no object x in $\{5, 16\}$ where `number_prime_factors(x)` > `number_of_divisors(x)`. The `number_of_divisors` bound is removed from the conjectures store as insignificant. The next change to the conjectures store comes after Iteration 9. The considered conjecture is "the number of prime factors of an integer x plus the number of its divisors is no more than $\pi(x)$". This is true for all known objects, and the bound passes the truth test. The bound gives a larger value than `number_of_divisors` for at least one known objects and is thus significant and added to the conjecture store. In fact `number_of_divisors` is no longer a significant bound and it is removed. At this point, for each object x, the maximum of the conjectured bounds equals the actual value of $\pi(x)$. No further bounds can be significant and the program stops.

Some of GRAFFITI's best-known conjectures are the following lower bounds for the independence number α of a graph (the maximum number of pairwise non-adjacent vertices in the graph, an NP-hard-to-compute graph invariant), and were made prior to the addition of the Dalmatian heuristic to that program.[5]

1. If G is a connected graph then $\bar{d} \leq \alpha$, where $\bar{d}$ is the *average distance* between distinct vertices of G. Both the average distance of a connected graph and the independence number of a graph are well-studied concepts. This conjecture was proved by Chung [36].
2. If G is a connected graph then $r \leq \alpha$ where r is the *radius* of G. This conjecture was originally proved by Erdős, Saks and Sós in [37]. Another proof, due to Fajtlowicz, follows from the characterization of graphs where its radius equals its independence number [38].
3. For any graph G, $R \leq \alpha$, where R is the *residue* of G (this is the number of zeros remaining after repeated application of the Havel–Hakimi procedure to the degree sequence of the graph). This conjecture was originally proved by Favaron, Mahéo and Saclé [39], and has since been reproved in the literature more than once. One nice proof is due to Griggs and Kleitman [40].

---

[5] The Dalmatian version of GRAFFITI could have also made these conjectures—they, in fact, were replicated by a Dalmatian-based conjecture-making program of the first author and Patrick Gaskill; see the blog at: independencenumber.wordpress.com for details.

Pre-Dalmatian versions of GRAFFITI were also used to generate some interesting number theory conjectures including, for instance, a formula for $\pi(x)$.

We now record some examples of GRAFFITI's Dalmatian heuristic conjectures in various domains.

1. (Graph theory, WoW #747) If G is a connected graph then $\bar{d} \leq \frac{b}{2}$, where b is the *bipartite number*, the order of a largest induced bipartite subgraph [1]. Here the objects are connected graphs, and the average distance and bipartite number are graph invariants. This conjecture generalizes Chung's theorem, mentioned earlier, and was recently proved by Hansen and collaborators [41].

2. (Geometry, WoW #738) For a vertex v of a triangle, $a \leq s$, where a is the length of the line segment which bisects v and extends to the opposite side and s is the length of the line segment from v to the midpoint of the opposite side. Here the objects are triangle vertices, and a and s are triangle vertex invariants. It is reasonable to believe that useful geometric knowledge awaits discovery. In fact, Coxeter has claimed that "geometry is developing as fast as any other kind of mathematics" [42]. Nevertheless, GRAFFITI's geometry conjectures did not initiate any research. This can be explained for various reasons; the most fundamental would be that this conjecture does not directly advance any existing research question, in particular, that there is neither any existing interest in upper bounds for the length of a line segment that bisects a vertex and extends to the opposite side, nor in lower bounds for the length of the line segment from a vertex to the midpoint of the opposite side.

3. (Chemistry, WoW #895) For any fullerene, the separator of the fullerene is no more than 1. Here the objects are fullerenes, and the separator is a fullerene invariant. Fullerenes are a recently-discovered (1985) form of carbon which include the 60-atom soccerball-shaped buckyball. Mathematically the structure of a fullerene is a trivalent planar graph with pentagonal and hexagonal faces. There are, for instance, 1812 mathematically possible 60-atom fullerenes but only one that appears in experiment [43]. Many other fullerenes with different atom counts have appeared in experiment as either hollow structures like C_{60}, endohedral structures, or which form tubes (which are technically incomplete fullerenes).

   The fullerene literature is now vast and includes a large number of papers connecting invariants of their mathematical structure to physical invariants. A mathematical fullerene invariant would include the difference between the largest and second-largest eigenvalue (or *separator*) of the symmetric matrix representing the bonding structure of its atoms. There is some empirical evidence connecting large separator values with fullerene stability [44,45]. GRAFFITI's conjecture was proved by Stevanović and Caporossi [46].

   It is worth noting that DeLaVina's GRAFFITI.PC program also made some conjectures related to the electronic structure of fullerenes; these are reported in [47, pp. 127–128].

   It might be said that these fullerene conjectures are more about graph theory than about chemistry. But graphs have been used to represent molecules since the 19th century—in a way that advances both chemists' understanding of their subject and their ability to predict chemical phenomenon [43]. A prototypical example is the Coulson–Rushbrooke Pairing Theorem [48] which can be found in both chemical and graph theoretic language: it can be stated either in terms of π-electron energy levels for alternant hydrocarbons or in terms of eigenvalues of bipartite graphs. In books such as Fowler and Manolopoulos' *Atlas of Fullerenes*, for instance, what might be counted as "chemistry" and what might be counted as "graph theory" can hardly be distinguished.

Fajtlowicz generated conjectures in domains other than graph theory largely to demonstrate that the heuristics that he invented were not limited to use in graph theory but were general ("domain independent").

4. 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 [49] and the numbers of generated structures have been compared to the implementation in [50]. 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 [49].

Importantly, the results we describe in the following sections are reproducible. Details about where to find the code and associated scripts are collected in the final section of this paper.

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.

5. *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.

6. *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 pre-defined 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 are desired, the user would enter 0 in the list of parameters to the C program.

A concrete example of the use of the program may be found in Appendix A. The *outputs* of the program are conjecture objects. These look like the statements given in the examples in the subsequent sections of this paper. In fact these conjectures are text representations of relationships between the invariants themselves, and have further useful features encoded in their methods. The program, by default, conjectures upper bounds for the chosen invariant. There are many other non-required user options. In particular, one option generates lower bound conjectures. The operators used in the expression are not user options. They are hard-coded in the program, since, unlike the functions computing the invariants, these operators are also needed in the C program. Additional operators can be added by making additions to the code. A user can exclude operators from being used. All conjectures reported here were made with the same set of operators; no code changes were made.

5. Matrix theory examples

Neither author is a matrix theorist, which is one reason that we chose matrix theory as a domain for experimentation. Sage has a number of built-in matrix invariants. We implemented other invariants (that we found in [51]) as Sage procedures; these function in the same way as Sage's built-in invariants.

The objects are symmetric matrices (which thus have the property that all of their eigenvalues are real). The invariants we used are the `determinant`, `nullity` (number of zero eigenvalues), `rank` (number of non-zero eigenvalues), `trace` (sum of the diagonal entries), `nrows` (number of rows of the matrix), `permanent`, `maximum_eigenvalue`, `minimum_eigenvalue`, `average_eigenvalue`, `number_of_distinct_eigenvalues`, `spectral_radius` (the difference between the largest and smallest eigenvalues), `ratio_min_max_absolute_eigenvalues` (the ratio of the absolute values of the largest and smallest eigenvalues), `sqrt_abs` (the square root of the sum of the absolute values of the entries of the matrix), `frobenius_norm` (the square root of the sum of the squares of the entries of the matrix),

Table 2
Upper bound conjectures for the determinant of a symmetric matrix.

1.	det(x)	≤	permanent(x)
2.	det(x)	≤	minimum_eigenvalue(x)*trace(x)
3.	det(x)	≤	maximum_eigenvalue(x)*trace(x)
4.	det(x)	≤	(rank(x) + 1)*spectral_radius(x)
5.	det(x)	≤	permanent(x)+max_column_sum(x)+1
6.	det(x)	≤	maximum(rank(x), minimum_eigenvalue(x)^2)
7.	det(x)	≤	maximum_eigenvalue(x)*minimum(minimum_eigenvalue(x), trace(x) + 1)
8.	det(x)	≤	minimum_eigenvalue(x)*minimum(trace(x), maximum_eigenvalue(x))
9.	det(x)	≤	maximum_eigenvalue(x)^l_inf_norm(x) + separator(x)
10.	det(x)	≤	trace(x)*average_eigenvalue(x) - permanent(x)
11.	det(x)	≤	(maximum_eigenvalue(x)+1)*minimum_eigenvalue(x)+frobenius_norm(x)

Table 3
Lower bound conjectures for the determinant of a symmetric matrix.

1.	det(x)	≥	minimum_eigenvalue(x)*separator(x)
2.	det(x)	≥	minimum(permanent(x), log(nullity(x)))
3.	det(x)	≥	-2*l_inf_norm(x)^nrows(x) + permanent(x)
4.	det(x)	≥	-(separator(x) - 1)*frobenius_norm(x) + permanent(x)
5.	det(x)	≥	-l_inf_norm(x)*frobenius_norm(x)
6.	det(x)	≥	minimum(rank(x)-1, minimum_eigenvalue(x)/nullity(x))
7.	det(x)	≥	-4*l_inf_norm(x)^2 + permanent(x)

Table 4
Upper bound conjectures for the absolute value of the determinant of a symmetric matrix.

1.	abs_det(x)	≤	log(nullity(x))^2
2.	abs_det(x)	≤	frobenius_norm(x)^2
3.	abs_det(x)	≤	frobenius_norm(x)^2/number_of_distinct_eigenvalues(x)
4.	abs_det(x)	≤	l_inf_norm(x)*frobenius_norm(x)
5.	abs_det(x)	≤	sqrt(maximum_eigenvalue(x)^2)*frobenius_norm(x)
6.	abs_det(x)	≤	4*minimum_eigenvalue(x)^2 + average_eigenvalue(x)
8.	abs_det(x)	≤	(spectral_radius(x)^2)^number_of_distinct_eigenvalues(x)
9.	abs_det(x)	≤	spectral_radius(x)^2/ratio_min_max_absolute_eigenvalues(x) + 1
10.	abs_det(x)	≤	(maximum_eigenvalue(x) - trace(x))*frobenius_norm(x)
11.	abs_det(x)	≤	maximum(separator(x)^2, maximum(permanent(x), l_inf_norm(x)))
12.	abs_det(x)	≤	-minimum_eigenvalue(x)*spectral_radius(x) + 1
13.	abs_det(x)	≤	1/2*minimum_eigenvalue(x)^2*spectral_radius(x)

Table 5
Lower bound conjectures for the absolute value of the determinant of a symmetric matrix.

1.	abs_det(x)	≥	average_eigenvalue(x) - 1
2.	abs_det(x)	≥	-rank(x)
3.	abs_det(x)	≥	l_inf_norm(x) - 1

max_column_sum (the maximum of the sum of the entries of each column), l_inf_norm (the maximum of the absolute value of the entries of the matrix), and the separator (the difference between the largest and second largest eigenvalues).

Bounds for the determinant of a matrix exist in the matrix theory literature. For instance, Hadamard's Inequality says that the determinant of a matrix is no more than the product of the Euclidean lengths of the vectors defined by its columns [51]. We presume that new bounds might also be of interest to researchers. The following conjectures should be taken as an example of what can be done with the program. Matrix theorists can generate conjectured bounds for any invariant that might advance their research.

The first round of conjectures for upper bounds for the determinant (det) of a symmetric matrix are included in Table 2. In general, we expect counterexamples of existing conjectures to be the best objects to input to a conjecture-making program. We find these ourselves, from experts, or by systematically generating matrices and testing whether each satisfied the conjecture or whether it was a counterexample.

We also generated a round of conjectured lower bounds for the determinant of a symmetric matrix. These can be found in Table 3. One further round of upper and lower bound conjectures for the absolute value of the determinant is included here. The upper bound conjectures are in Table 4. The lower bound conjectures for the absolute value of the determinant of a matrix are in Table 5. The second conjecture is, of course, trivially true; nevertheless, at the point that it was made it had to have given a better bound for the absolute value of the determinant of some object than the bound in the first conjecture.

Table 6
Lower bound conjectures for Goldbach(x) (1st round).

1.	Goldbach(x)	≥	1/digits10(x)
2.	Goldbach(x)	≥	digits10(x) - 1

Table 7
Lower bound conjectures for Goldbach(x) (2nd round).

1.	Goldbach(x)	≥	euler_phi(x)/prime_pi(x)
2.	Goldbach(x)	≥	-euler_phi(x) + prime_pi(x)
3.	Goldbach(x)	≥	-count_divisors(x) + digits2(x)
4.	Goldbach(x)	≥	1/2*sqrt(1/2)*sqrt(euler_phi(x))
5.	Goldbach(x)	≥	-1/2*mertens(x)

While one use for conjectured bounds for an invariant is as statements for mathematical investigation, another possible use is for the heuristic estimation of an invariant value. An estimate for the value of the determinant of a matrix can be made by using either of the minimum of the values of the conjectured upper bounds for the determinant, or the maximum of the values of the conjectured lower bounds. While the determinant of a matrix is efficiently computable and an estimate may not be of any practical interest, there are other hard-to-compute invariants—for example, the independence number of a graph—where estimates generated from efficiently computable conjectured bounds may be of interest. Another case where conjecture-driven invariant estimates might be of use is in characterizing a class of objects. An example will be discussed in connection with the game Chomp.

6. Number theory conjectures

We also chose number theory as a domain for experimentation because neither author has any special knowledge of this subject. We were aware of Goldbach's Conjecture, interest in the distribution of the prime numbers and the Prime Number Theorem, and the Riemann Hypothesis. While we did not expect to make any contribution to this classical and well-researched area of mathematics, we did think that our experiments could be of interest.

Goldbach, in a 1742 letter to Euler, conjectured that every even integer ($n > 2$) is the sum of two primes [52]. The conjecture has been verified by computer for all integers up to at least $4 \cdot 10^{14}$ [53], and there are a large number of partial results and continuing interest. Let Goldbach(x) be the number of representations of x as a sum of two primes (so Goldbach(6) = 1 and Goldbach(8) = 2). Clearly Goldbach's Conjecture is true if and only if Goldbach(x) > 0 for even $x > 2$. Thus, conjectured lower bounds for Goldbach(x) are of potential utility to researchers: they could yield new avenues of investigation.

The objects here are even integers, and the invariants are integer invariants. The first run of conjectures for Goldbach(x) involved the invariants prime_pi (the number of primes $\leq x$), euler_phi (the number of integers $\leq x$ which are relatively prime to x), number (which returns the number x itself), digits10 (the number of digits in the base-10 representation of x), digits2 (the number of digits in the base-2 representation of x), sigma (the sum of all divisors of x), count_divisors (the number of divisors of x), next_prime (the smallest prime greater than x), previous_prime (the largest prime smaller than x), and count_quadratic_residues (the number of quadratic residues of x). Most of these invariants were included merely because they were built-in Sage functions.

Table 6 records the first round of Goldbach(x) conjectures. Conjecture 1 is curious as the bound goes to 0 as x gets large, and the bound is of little predictive use. Nevertheless, the only object initially known to the program was the integer 4, and for this object the lower bound exactly predicts the true value of 1 for Goldbach(4). The second conjecture, which is much stronger in general, fails to predict Goldbach(4) and thus does not supersede the first conjecture (the first conjecture is "stronger" for x = 4). Both conjectures were tested and were true for values of x up to 1,000,000. No counterexample was found to the second conjecture and, thus, no further conjectures could be added to the list of conjectures. The second conjecture is, in general, stronger than Goldbach's Conjecture and, if true, would imply it.

In order to generate different conjectures, we removed digits10 as an invariant, and added some new invariants: mertens (the sum of the values of the Möbius function for the integers no more than x [54, p. 36]), li (the logarithmic integral with lower bound 0), zeta (the Riemann zeta function), reciprocal_prime_sum (the sum of the reciprocals of the primes up to x), max_prime_divisor (the largest prime divisor of x), and prime_product (the product of all primes no more than x). The second run of the program produced the conjectures in Table 7.

The distribution of the prime numbers has been of interest at least since the time of Gauss, who famously conjectured that $\pi(x)$, the number of primes no more than x, is asymptotic to $\frac{x}{\log(x)}$. This is now the Prime Number Theorem (PNT), which was proved independently by Hadamard and Vallée–Poussin in 1896 [52]. prime_pi(x) is the Sage implementation of $\pi(x)$. There is an interest in getting good explicit estimates for $\pi(x)$. One bound due to Rosser [55], says that, for $x \geq 55$,

$$\frac{x}{\log x + 2} < \pi(x) < \frac{x}{\log x - 4}$$

Table 8
Upper bound conjectures for $\pi(x)$ (1st round).

1.	prime_pi(x)	≤	previous_prime(x)
2.	prime_pi(x)	≤	(number(x) - 1)/digits10(x)
3.	prime_pi(x)	≤	1/2*sigma(x)
4.	prime_pi(x)	≤	digits10(x) + euler_phi(x)
5.	prime_pi(x)	≤	maximum(count_quadratic_residues(x), euler_phi(x) + 1)
6.	prime_pi(x)	≤	maximum(euler_phi(x), 1/2*number(x))
7.	prime_pi(x)	≤	digits2(x)^2 + digits10(x)
8.	prime_pi(x)	≤	count_quadratic_residues(x) + 1/2*euler_phi(x) - 1
9.	prime_pi(x)	≤	(next_prime(x) + number(x) - 1)/digits2(x)
10.	prime_pi(x)	≤	2*sigma(x)/digits2(x)
11.	prime_pi(x)	≤	maximum(euler_phi(x), 2*count_divisors(x))
12.	prime_pi(x)	≤	2*sqrt(previous_prime(x)) + count_quadratic_residues(x)
13.	prime_pi(x)	≤	(previous_prime(x) + euler_phi(x))/log(number(x))
14.	prime_pi(x)	≤	(digits10(x)^2 + 1)^2
15.	prime_pi(x)	≤	4*count_quadratic_residues(x) - 2*digits2(x)
16.	prime_pi(x)	≤	maximum(1/2*previous_prime(x), digits10(x) + digits2(x))
17.	prime_pi(x)	≤	next_prime(x)/(log(number(x)) - 1)
18.	prime_pi(x)	≤	sigma(x)*count_divisors(x)/digits2(x)
19.	prime_pi(x)	≤	2*(number(x) + 1)/digits2(x)
20.	prime_pi(x)	≤	1/2*digits2(x) + 1/4*next_prime(x)
21.	prime_pi(x)	≤	maximum(digits2(x)^2, euler_phi(x) - count_divisors(x))
22.	prime_pi(x)	≤	maximum(euler_phi(x) - 1, 2*count_divisors(x))
23.	prime_pi(x)	≤	2*(next_prime(x) - 1)/digits2(x)
24.	prime_pi(x)	≤	log(10)*previous_prime(x)/(log(digits10(x))*digits2(x))
25.	prime_pi(x)	≤	2*count_quadratic_residues(x) + digits10(x) + 2
26.	prime_pi(x)	≤	sigma(x)/(log(number(x)) - 1)
27.	prime_pi(x)	≤	log(digits2(x) + 1)*count_quadratic_residues(x)
28.	prime_pi(x)	≤	previous_prime(x)/log(1/2*count_quadratic_residues(x))
29.	prime_pi(x)	≤	1/4*count_divisors(x)^2 + count_quadratic_residues(x)
30.	prime_pi(x)	≤	sqrt(1/count_divisors(x))*next_prime(x)
31.	prime_pi(x)	≤	(log(count_divisors(x))/log(10) + 1)*count_quadratic_residues(x)
32.	prime_pi(x)	≤	next_prime(x)/(digits10(x) + 1) + 1
33.	prime_pi(x)	≤	4*sqrt(previous_prime(x)) + 1
34.	prime_pi(x)	≤	(euler_phi(x) + number(x))/log(next_prime(x))
35.	prime_pi(x)	≤	(sigma(x) + digits2(x))/(digits10(x) + 1)
36.	prime_pi(x)	≤	-previous_prime(x) + 1/2*count_quadratic_residues(x) + sigma(x)
37.	prime_pi(x)	≤	count_divisors(x)^digits10(x) + 1/2*count_quadratic_residues(x)
38.	prime_pi(x)	≤	sigma(x)*count_divisors(x)/digits2(x)
39.	prime_pi(x)	≤	1/2*count_quadratic_residues(x) - euler_phi(x) + next_prime(x)
40.	prime_pi(x)	≤	digits2(x)^2 - previous_prime(x) + next_prime(x)
41.	prime_pi(x)	≤	(previous_prime(x) + digits2(x))/sqrt(count_divisors(x))
42.	prime_pi(x)	≤	maximum(2*digits2(x), previous_prime(x) - count_quadratic_residues(x))
43.	prime_pi(x)	≤	maximum(euler_phi(x) - 1, count_quadratic_residues(x) + digits2(x))
44.	prime_pi(x)	≤	digits2(x)*sqrt(count_divisors(x)) + count_quadratic_residues(x)
45.	prime_pi(x)	≤	log(previous_prime(x))*count_divisors(x)/log(10) + count_quadratic_residues(x)
46.	prime_pi(x)	≤	maximum(count_quadratic_residues(x), 1/2*sigma(x)/digits10(x))
47.	prime_pi(x)	≤	sqrt(count_divisors(x))^digits10(x) + count_quadratic_residues(x)
48.	prime_pi(x)	≤	-(digits10(x) - 2*next_prime(x))/digits2(x)
49.	prime_pi(x)	≤	-(sqrt(digits10(x)) - count_divisors(x))*number(x)

Other well-known bounds of Rosser and Schoenfeld can be found in [56].

Conjectures from the first run of the program are recorded in Table 8. The invariants used in these are the same as those used in the first run of conjectures for Goldbach(x). By the design of the program, we know that the truth of these conjectures was tested for only 49 integers (all less than 440). Conjectures 2, 13 and 26 from Table 8 seemed of possible interest as they are similar in form to the PNT. We further tested the truth of these three conjectures for all integers no more than $x = 1,000,000$. Conjecture 26 is false for $x = 467$. Conjectures 2 and 13 are true for all tested values of x.

Lastly, we generated some conjectures that could be related to the Riemann Hypothesis. The best-known version of the conjecture, and the one posed by Riemann, is a claim about the zeros of the Riemann zeta function. Von Koch showed that the Riemann hypothesis is equivalent to the statement that $|\pi(x) - Li(x)| \leq \sqrt{x} \cdot \ln(x)$ for $x \geq 2.01$, where $Li(x) = \int_2^x \frac{1}{\ln(t)} dt$ is the (offset) logarithmic integral (so $Li(x) = li(x) - li(2)$) [54, p. 37]. Thus we expect bounds for $|\pi(x) - Li(x)|$ to be of interest to researchers.

In our initial run, we encountered the Sorcerer's Apprentice Problem. Starting with a single (integer) object the program generated conjectures, found counterexamples, added these to the store of objects, and repeated the process. In making repeated rounds, the program soon produced 200 counterexamples and 71 corresponding conjectures and was stopped (the first 25 conjectures are in Table 9). The list had become too long and of less and less potential interest: experts can't be expected to consider 71 conjectures with as much interest as they might have for a mere handful of conjectures.

Table 9
Upper bound conjectures for $|\pi(x) - Li(x)|$ (2nd round).

1.	$\|\pi(x) - Li(x)\|$	$\leq$	`sqrt(number(x))/reciprocal_prime_sum(x) - 1`
2.	$\|\pi(x) - Li(x)\|$	$\leq$	`reciprocal_prime_sum(x)**digits10(x) + 1`
3.	$\|\pi(x) - Li(x)\|$	$\leq$	`sqrt(number(x) + 1) + mertens(x)`
4.	$\|\pi(x) - Li(x)\|$	$\leq$	`sqrt(sigma(x)/(count_divisors(x) + 1))`
5.	$\|\pi(x) - Li(x)\|$	$\leq$	`log(next_prime(x))*sqrt(digits2(x))/log(10)`
6.	$\|\pi(x) - Li(x)\|$	$\leq$	`mertens(x)/digits10(x) + digits2(x)`
7.	$\|\pi(x) - Li(x)\|$	$\leq$	`(digits2(x)/digits10(x))**reciprocal_prime_sum(x)`
8.	$\|\pi(x) - Li(x)\|$	$\leq$	`sqrt(sigma(x)) + mertens(x)`
9.	$\|\pi(x) - Li(x)\|$	$\leq$	`reciprocal_prime_sum(x)**2 + log(count_quadratic_residues(x))`
10.	$\|\pi(x) - Li(x)\|$	$\leq$	`-euler_phi(x)/(reciprocal_prime_sum(x) - digits2(x))`
11.	$\|\pi(x) - Li(x)\|$	$\leq$	`number(x)/log(next_prime(x))**2`
12.	$\|\pi(x) - Li(x)\|$	$\leq$	`sqrt(count_quadratic_residues(x)*sqrt(reciprocal_prime_sum(x)))`
13.	$\|\pi(x) - Li(x)\|$	$\leq$	`sqrt(1/2)*sqrt(sigma(x)) - mertens(x)`
14.	$\|\pi(x) - Li(x)\|$	$\leq$	`(log(number(x))/log(10))**reciprocal_prime_sum(x) + 1`
15.	$\|\pi(x) - Li(x)\|$	$\leq$	`sqrt(sigma(x)/digits10(x))*sqrt(1/2)`
16.	$\|\pi(x) - Li(x)\|$	$\leq$	`1/2*max(max_prime_divisor(x), 1/2*euler_phi(x))`
17.	$\|\pi(x) - Li(x)\|$	$\leq$	`sqrt(euler_phi(x)) - log(max_prime_divisor(x))/log(10)`
18.	$\|\pi(x) - Li(x)\|$	$\leq$	`-log(2*count_divisors(x))/log(10) + digits2(x)`
19.	$\|\pi(x) - Li(x)\|$	$\leq$	`max(digits10(x)**2, 1/2*count_divisors(x))`
20.	$\|\pi(x) - Li(x)\|$	$\leq$	`sigma(x)**(sqrt(reciprocal_prime_sum(x)) - 1)`
21.	$\|\pi(x) - Li(x)\|$	$\leq$	`1/2*sqrt(euler_phi(x)) + 1/2*max_prime_divisor(x)`
22.	$\|\pi(x) - Li(x)\|$	$\leq$	`sqrt(sigma(x)) + mertens(x)`
23.	$\|\pi(x) - Li(x)\|$	$\leq$	`sqrt(1/2*euler_phi(x) + digits2(x))`
24.	$\|\pi(x) - Li(x)\|$	$\leq$	`max(count_divisors(x), sqrt(count_quadratic_residues(x))) - 1`
25.	$\|\pi(x) - Li(x)\|$	$\leq$	`-1/2*log(prime_product(x)) + 1/2*sigma(x)`

Table 10
Upper bound conjectures for $|\pi(x) - Li(x)|$ that improve on the Von Koch bound for $x \leq 1{,}000{,}000$.

1.	$\|\pi(x) - Li(x)\|$	$\leq$	`-digits10(x)^(1/4) + digits2(x)`
2.	$\|\pi(x) - Li(x)\|$	$\leq$	`sqrt(number(x)) - log(euler_phi(x))`
3.	$\|\pi(x) - Li(x)\|$	$\leq$	`maximum(digits10(x), 1/2*sqrt(previous_prime(x)))`

The conjectures that we really wanted were ones that would be of possible use in proving the Riemann Hypothesis. In light of this, we filtered the conjectured bounds for the ones that were no more than $\sqrt{x} \cdot \ln(x)$ for all integers x between 3 and 1,000,000. The produced conjectures satisfying this additional condition are in Table 10.

7. Bounds for the graph theoretic domination number

Both authors are graph theorists. Our interests and research specialties include independence number theory, graph generation, chemical graph theory, and algorithm design; neither of us is an expert in the theory of dominating sets and domination number of a graph.

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 [57]. 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 [58]. 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 [59,60], and the *residue* is a degree sequence lower bound for the independence number introduced by Fajtlowicz [39].

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* [61] to generate all graphs up to some (small) specified order in a loop to automatedly find counterexamples to generated conjectures and, thus, automatically improve the produced conjectures. In our run

Table 11
Upper bound conjectures for the domination number of a graph.

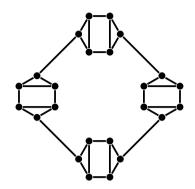|			
1.	`domination_number(x)`	≤	`fractional_independence_number(x)`
2.	`domination_number(x)`	≤	`annihilation_number(x)`
3.	`domination_number(x)`	≤	`residue(x) + 1`

Fig. 1. A counterexample to Conjecture 3 in Table 11.

Table 12
Upper bound conjectures for the domination number of a graph (1st run).

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

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 11. The first two we knew to be true. The third is false: Ryan Pepper found a 24 vertex counterexample; this is the graph in Fig. 1.

After adding Pepper's counterexample, we generated another run of upper bound domination conjectures. These are in Table 12. Stephen Hedetniemi, a co-author of the standard reference on domination [57], points out that the second of these conjectures is false for K_1 and K_2—we 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. The graph in Fig. 2 disproves these: it has 201 vertices, domination number of 100, girth of 3, maximum degree of 100, average distance of 2.97, diameter of 4, and radius of 2.

We added Hedetniemi's counterexamples as objects to the program and generated a second round of conjectures for upper bounds for the domination number of a graph; the results are in Table 13. Several of these conjectures reappeared in the second round. All of these conjectures other than the first remain open.

Finally, we generated a round of conjectured lower bounds for the domination number of a graph. The results are in Table 14. The first conjecture is false: Pepper points out that large enough cycles are counterexamples. The second and third conjectures are curious as they are trivially true; but, at some point in the conjecture-making process, there must have been graphs for which these bounds gave a larger predicted domination number than the previously conjectured bounds.
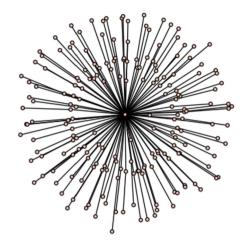
Fig. 2. A counterexample to several conjectures in Table 12.

Table 13
Upper bound conjectures for the domination number of a graph (2nd run).

1.	domination_number(x)	≤	1/2*order(x)
2.	domination_number(x)	≤	size(x) - 1
3.	domination_number(x)	≤	2*max_degree(x) + 2
4.	domination_number(x)	≤	residue(x)^2
5.	domination_number(x)	≤	order(x) - max_degree(x)
6.	domination_number(x)	≤	diameter(x) + max_degree(x) - 1
7.	domination_number(x)	≤	2*residue(x) - 1
8.	domination_number(x)	≤	max(residue(x), diameter(x) + 1)
9.	domination_number(x)	≤	max(max_degree(x), 2*diameter(x) - min_degree(x))

Table 14
Lower bound conjectures for the domination number of a graph (1st run).

1.	domination_number(x)	≥	radius(x) - 1
2.	domination_number(x)	≥	1/average_distance(x)
3.	domination_number(x)	≥	1/radius(x)

8. Lower bounds for the graph-theoretic independence number

The *independence number* of a graph (also called the *stability number*, *vertex packing number* and *node packing number*) is the largest number of mutually pairwise non-adjacent vertices in the graph. It is NP-hard to compute [62], of both practical and theoretical interest, and has a large literature. There are at least 50 published bounds for this graph invariant. In this case our goal was to generate conjectured lower bounds for the independence number which are not consequences of existing (proved) lower bounds. It is practically impossible to compute the independence number of arbitrary graphs of even moderate size (say, with 2000 vertices[6]).

In this case the authors are specialists. The first author has written papers on the theory of maximum independent sets, the computation of this number, and applications [63,64,60,65,45]. But a similar investigation could have been carried out by a non-specialist who had the collected published facts that we make use of. Special use was made of this knowledge only to find counterexamples to conjectures; if finding counterexamples could be automated then this investigation could have been done entirely without expert knowledge.

In the previously reported experiments, no existing bounds were included in the program—and, thus, the program could make "rediscoveries." Mazur later suggested including known bounds—theoretical knowledge—in the program to keep the program from making conjectures that were implied by existing theory. This was easy to implement: the conjectures store can simple be seeded with these known bounds. In this way, the program only produces conjectures that are mathematically significant in a precise sense: they will give better invariant value predictions for some objects than any known bounds will. The known bounds that we included in these experiments are the following: the aforementioned *radius* and *residue*, Fajtlowicz's max-even-minus-min-horizontal bound (the maximum, over all vertices *v*, of the number of vertices minus the number of edges for the subgraph induced by the vertices at even distance from *v*), together with a bound due to Angel, Campigotto and Laforest [66].

---

[6] Neil Sloan maintains a list of some of these that have been beyond the abilities of experts at: neilsloane.com/doc/graphs.html.
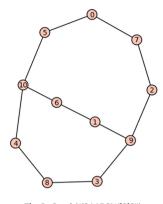
Fig. 3. Graph("J?AAD?W[?[?").

Table 15
Lower bound conjectures for the independence number of a graph.

1.	independence_number(x)	≥	density(x) + lovasz_theta(x) - min_degree(x) + 1
2.	independence_number(x)	≥	card_periphery(x) - max_degree(x) - residue(x)

The upper bound theory for the independence number is surprisingly good: there is an efficiently computable upper bound for the independence number of a graph which can give very good estimates for the true value of this invariant: this is the famous Lovász number ϑ of a graph. Lovász's original definition was in terms of orthogonal representations of the graph [67]. There are now known to be several equivalent definitions [68]. All these definitions are all relatively technical and not worth detailing here. One definition is as the optimal value of a semidefinite program—and solving semidefinite programs is something that can be done efficiently [69]. We have computed both the Lovász number and the independence number for all graphs with up to 10 vertices (there are more than 12 million simple graphs with 10 vertices [70]). For all of these graphs, the floor of the Lovász number equals the independence number; that is, the Lovász number exactly predicts the independence number.

No similar lower bound exists. For small graphs the residue can be very good. But, as the order of the graph increases, the residue becomes less and less good. Our goal was to find new lower bounds that would advance this theory. While it is unlikely that there is any single lower bound that will be as predictive as the Lovász number upper bound, it is possible that, with enough lower bounds, at least one of them may be predictive for a given graph. We conducted two investigations, the first was for small "hard" graphs, and the second was for random graphs.

In our first investigation we used only *independence irreducible* graphs [64] as examples for the program. These graphs admit various definitions. The significance of these graphs is that the calculation of the independence number of any graph can be efficiently reduced to the calculation of the independence number of an independence irreducible subgraph. In this sense these are the "hardest" graphs for independence number computations.

Here we included all of the efficiently computable graph invariants that are either built-in to Sage or that we have coded (as Sage procedures). So this investigation did not include domination_number but did include, for instance, lovasz_theta. The complete list is: independence_number, card_center, Graph.connected_components_number, cycle_space_dimension, card_periphery, Graph.density, Graph.average_distance, Graph.diameter, Graph.radius, Graph.girth, Graph.order, Graph.size, Graph.szeged_index, min_degree, Graph.wiener_index, max_degree, Graph.average_degree, matching_number, residue, annihilation_number, lovasz_theta, and cvetkovic.

Here we do not report all the conjectures made by the program but only those that attracted the first author due either to their simplicity or their relation to known bounds. The first of these is the conjecture that, for independence irreducible graphs, independence_number ≥ min(girth-1, cvetkovic). cvetkovic is the *Cvetković bound*, the minimum of the number of non-negative and non-positive eigenvalues of the graph. It is an upper bound of the independence number. The fact that it showed up in a simple formula for a conjectured lower bound is surprising. The conjecture was verified for all graphs of up to ten vertices. But there is a counterexample with eleven vertices: this is the graph in Fig. 3.

The next conjecture is that independence_number ≥ 2*matching_number-2*maximum_degree for all independence irreducible graphs. This conjecture was also verified for all graphs with up to ten vertices. But here there is a counterexample with fourteen vertices, namely the graph formed by putting a single edge from any vertex in one 7-cycle (a cycle graph on seven vertices) to any vertex on a second 7-cycle.

The two conjectures in Table 15, which were also verified for all graphs with up to ten vertices, remain open.

In the previous run of the program, potential counterexamples were searched for by systematically generating all small graphs using McKay's *geng*. Counterexamples were then added as objects to the program, and the process was iterated. In the next run of the program we did not use *geng*. We wanted to use larger graphs as potential counterexamples and here systematic generation is impossible: there are, for instance, approximately 10^{65} graphs of order 25. We chose random selec-

Table 16
Lower bound conjectures for the independence number of a graph.

1.	`independence_number(x)`	$\geq$	`min{girth(x), matching_number(x)-1}`
2.	`independence_number(x)`	$\geq$	`periphery(x)-2*maximum_degree(x)`
3.	`independence_number(x)`	$\geq$	`min{edges(x)-order(x)+1,diameter(x)}`

tions of graphs from orders 10 to 25 in our search for counterexamples. These were generated, essentially, by choosing the order of the graph and, for each pair of vertices, flipping a coin to determine whether or not to include an edge between the vertices (the theory of *random graphs* is large and dates back to the 1950s [71]). Three of these conjectures are in Table 16. They should be interpreted as being true for random graphs. The first and third conjectures in Table 16 are interesting because they, in some sense, generalize the radius lower bound for the independence number of a graph (mentioned earlier, following a conjecture of GRAFFITI). The second conjecture involves the `periphery`, which is the cardinality of the set of vertices of maximum eccentricity.

9. Chomp

A conjecture-making program can be used in the design of a game-playing program. We describe how this might be done, and an initial experiment, for the game of Chomp. Chomp is an impartial two-player perfect information game that terminates in a finite number of moves; thus it has a winning strategy [72]. The game is played on a rectangular board which is, at the beginning, completely covered with "cookies". The upper left corner of the board contains a "poisoned cookie". Which ever player eats the poison cookie loses. A move consists of a player choosing a square with a cookie on it, removing that cookie and all cookies to the right of it or below it and any other cookie which is both to the right and below it.

There are board positions from which, if the next player to move *N* played perfectly, *N* would be guaranteed to win, regardless of what future moves the previous player to play *P* makes. The winning strategy for *N* is known in special cases, for instance, if the cookies form an "L", or if there are only one or two rows of cookies. A winning strategy for *N* in the general case is not known. Given a board position, what is the best move for *N* to make in order to secure an eventual victory? A position may be theoretically "solvable" despite the fact that no human (nor machine) knows the solution—it is beyond current human and machine powers. *N*'s goal is to get from the current board position to one that is known to be solvable.

Our idea is use known winning positions (where *N* has a known winning strategy) as objects, and generate conjectures about these positions to use in the choice of a move. For Chomp the game board can easily be represented as a 0–1 matrix with 1s in the entries representing squares of the board with cookies on them. Various invariants can then be defined. We defined only invariants that are Chomp-specific; that is, we did not use any matrix invariants such as the largest singular value, that did not have any obvious interpretation in terms of game play. "Chomp conjectures" involving only natural Chomp invariants might be provable—in terms of game strategy—and initiate a Chomp Theory. Our ideal Chomp Theory would consist of some number of invariant-relation statements that would all be satisfied for any winning position and which would not all be satisfied for a losing position.

The invariants we used are: `cookie_rank` (the number of different rows with a different number of 1s, equals the matrix rank), `duplicate_columns` (the number of duplicated columns), `duplicate_rows` (the number of duplicated rows), `first_two_columns_difference` (the difference between the number of 1s in the first and second columns), `first_two_rows_difference` (the difference between the number of 1s in the first and second rows), `possible_cookies` (the initial number of 1s at the beginning of the game), `number_of_cookies` (the number of 1s in the matrix), `L_difference` (difference between the length of the longest row and longest column), `diagonal_cookies` (the number of 1s on the diagonal of the matrix), `max_column_cookies` (the number of 1s in the leftmost column of the matrix), `max_row_cookies` (the number of 1s in the topmost row of the matrix), and `inside` (the number of 1s not in the leftmost column or topmost row).

The objects that we used were 26 mostly simple winning game positions derived from Gale's original paper. We then generated conjectured upper and lower bounds for each of the 12 invariants. A possible game position would either satisfy these "winning position conjectures" or it would not: if every conjecture were true then any winning position would satisfy *all* of the conjectures. At any moment in the game *N* will have a number of possible moves. The move heuristic that we used was to choose the move yielding a board position that satisfied the *fewest* number of the conjectures—on the theory that this move would yield a position that was least likely to be a winning position for *P*. We did not implement any look-ahead in our move heuristic.

A sample of the generated conjectures—all the upper bound conjectures for the number of cookies on the diagonal of the game board—are recorded in Table 17. To the extent that all of the conjectures are true, it may be that they completely characterize winning game positions. In this case a player, human or robot, basing their moves on these conjectures can play a maximally intelligent game of Chomp.

Table 17
Upper bound conjectures for the number of cookies on the diagonal of a game position.

1.	diagonal_cookies(x)	≤	possible_cookies(x)
2.	diagonal_cookies(x)	≤	max_column_cookies(x)
3.	diagonal_cookies(x)	≤	L_difference(x) + 1
4.	diagonal_cookies(x)	≤	duplicate_rows(x)
5.	diagonal_cookies(x)	≤	cookie_rank(x) - 1
6.	diagonal_cookies(x)	≤	(1/cookie_rank(x))
7.	diagonal_cookies(x)	≤	sqrt(L_difference(x))
8.	diagonal_cookies(x)	≤	min(duplicate_columns(x), max_column_cookies(x))
9.	diagonal_cookies(x)	≤	cookie_rank(x) - first_two_columns_difference(x)
10.	diagonal_cookies(x)	≤	cookie_rank(x) - first_two_rows_difference(x)
11.	diagonal_cookies(x)	≤	-cookie_rank(x) + first_two_rows_difference(x)

10. Future work

These preliminary results show that a conjecture-making program based on the Dalmatian heuristic can be of use to researchers in their investigations of bounds of real number invariants of objects. Our example of a conjecture-based Chomp playing program suggests how an automated conjecture-making functionality might be used in the design of other intelligent behaviors.

Researchers who use our program may be able to improve their results in various ways. Some possibilities include the following.

1. Add more invariants. This is not an issue in the design of our program but, rather, in its use. We used relatively small numbers of invariants. In contrast, DeLaVina's GRAFFITI.PC included more than 100 invariants when it was originally developed in 2001 [2], and which now has many more.
2. Vary the counterexample-finding process. Instead of systematically generating all possible small counterexamples, or choosing random examples of various size objects, it might be useful to choose examples that are extremal with respect to one of the invariants in the formula for a conjectured bound. Hansen and Caporossi's AGX, for instance, is able to generate these examples.
3. Add existing bounds (ones proved in the literature) and conjectures to the program. In this way, the program could only produce conjectures that for which there is at least one example where the produced conjecture gives a better bound than any known theorem or conjecture. In Fajtlowicz's terms, the produced conjectures would be "informative" or "significant" compared to existing theory. We did not do this in our initial experiments but did use this idea successfully for our independence number conjectures.
4. Use a *property-relations* conjecture-making program. Generated counterexamples may have certain properties. If these commonalities can be identified, it may be possible to refine many conjectures which are "almost true". The generalization of the Dalmatian heuristic to the design of a program that makes conjectures about relations of properties of objects is explained below.

Lastly we discuss an idea for extending the automation of invariant-relation conjectures to the automation of property-relation conjectures. The Dalmatian heuristic, as described, is used to make conjectures about relations between the invariants of an object. Conjectures between the properties of an object may also be of interest. A property is a condition that an object does or does not have. An integer is perfect if it equals the sum of its proper divisors. "Being perfect" is an integer property: any given integer does or does not have this property. ("Being wet" is not an integer property.) An example of a property-relation conjecture is: If an integer is perfect then it is even. This can also be stated as "Being perfect is a sufficient condition for being even" or, "Being even is a necessary condition for being perfect."

The analogues of upper or lower bounds for an invariant of interest are necessary or sufficient conditions for a property of interest. Let P be the property that an integer is perfect. If sufficient conditions for an integer to have this property are desired, a conjecture-making program would need to produce property-expressions $Q_1, Q_2, \ldots$, and statements of the form, "If an integer has property Q_i then it has property P" (or, more simply, "If Q_i then P"). If necessary conditions are desired then the program would need to produce statements of the form, "If an integer has property P then it has property Q_i."

Let $\mathcal{O}_1, \ldots, \mathcal{O}_n$ be examples of objects of a given type. Let $Q_1, \ldots, Q_k$ be properties. And let P be a property for which conjectured necessary or sufficient conditions are of interest. If the objects are the integers $G_1, \ldots, G_n$, and P is the property "is perfect" then $P(G_i)$ would be True if G_i is perfect and False if G_i is not perfect.

An unlimited stream of boolean functions of the invariants can then be formed: $Q_1 \wedge Q_2$, $\neg Q_1$, $Q_1 \vee Q_3$, $(Q_2 \wedge Q_4) \vee Q_3$, etc. This stream can be produced in any way at all. These expressions can then be used to form conjectured necessary or sufficient conditions for P. If we are interested in necessary conditions for P, say, we can form the statements $P \Rightarrow Q_1 \wedge Q_2$, $P \Rightarrow \neg Q_1$, $P \Rightarrow Q_1 \vee Q_3$, $P \Rightarrow (Q_2 \wedge Q_4) \vee Q_3$, etc. These statements can be interpreted as being true for all the objects of the given type. That is, the statement $P \Rightarrow Q_1 \wedge Q_2$ can be interpreted as, "For every object $\mathcal{O}$, $P(\mathcal{O}) \Rightarrow Q_1(\mathcal{O}) \wedge Q_2(\mathcal{O})$." A conjectured necessary condition Q is only added to the database of conjectures if the property passes the following two tests.

1. (*Truth test.*) The candidate conjecture $P \Rightarrow Q$ 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 $\neg Q(\mathcal{O}) \wedge (Q_1(\mathcal{O}) \wedge \ldots \wedge Q_r(\mathcal{O}))$, where $Q_1, \ldots, Q_r$ are the currently stored conjectures. That is, the candidate conjecture would give a better necessary condition for $P(\mathcal{O})$ than any previously conjectured necessary condition.

If we are interested in sufficient conditions for P we can form the statements $Q_1 \wedge Q_2 \Rightarrow P$, $\neg Q_1 \Rightarrow P$, $Q_1 \vee Q_3 \Rightarrow P$, $(Q_2 \wedge Q_4) \vee Q_3 \Rightarrow P$, etc. These statements can be interpreted as being true for all the objects of the given type. That is, the statement $Q_1 \wedge Q_2 \Rightarrow P$ can be interpreted as, "For every object $\mathcal{O}$, $Q_1(\mathcal{O}) \wedge Q_2(\mathcal{O}) \Rightarrow P(\mathcal{O})$." A conjectured sufficient condition Q is only added to the database of conjectures if the property passes the *Truth* and *Significance* tests. In this case the significance test would be as follows: Check that there is an object $\mathcal{O} \in \{\mathcal{O}_1, \ldots, \mathcal{O}_n\}$ such that $Q(\mathcal{O}) \wedge \neg(Q_1(\mathcal{O}) \wedge \ldots \wedge Q_r(\mathcal{O}))$, where $Q_1, \ldots, Q_r$ are the currently stored conjectures. That is, the candidate conjecture would give a better sufficient condition for $P(\mathcal{O})$ than any previously conjectured sufficient condition.

Another way to think about property-relation conjectures is in terms of the sets of objects that have some property. Let $\mathcal{P}$ be the set of objects that have property P. Necessary conditions for membership in $\mathcal{P}$ define a super-class $\mathcal{N}$ of $\mathcal{P}$. What is wanted are conjectures that make this super-class smaller and smaller. So a conjectured necessary condition Q is informative if, together with the previous conjectures, it defines a smaller super-class $\mathcal{N}'$ with $\mathcal{P} \subseteq \mathcal{N}' \subset \mathcal{N}$.

Similarly, sufficient conditions for membership in $\mathcal{P}$ define a sub-class $\mathcal{S}$ of $\mathcal{P}$. What is wanted here are conjectures that make this sub-class larger and larger. So a conjectured sufficient condition Q is informative if, together with the previous conjectures, it defines a larger sub-class $\mathcal{S}'$ with $\mathcal{S} \subset \mathcal{S}' \subseteq \mathcal{P}$.

The first author described a Dalmatian-style necessary condition heuristic in [13]. DeLaVina and Waller described and implemented a Dalmatian-style sufficient condition heuristic in [73], that they call "Sophie". The Sophie version of Graf-fiti.pc has produced some useful conjectures including the following.

1. (WoW2 #196a) For any connected graph G, if $\alpha(G) = r(G)$ then G has a Hamiltonian path [73], where α is the independence number of the graph, and r is the radius. A Hamiltonian path in a graph is a path which visits each vertex exactly once. Here the objects are graphs, and the properties are "has equal independence number and radius" and "has a Hamiltonian path". Sufficient conditions for a graph having a Hamiltonian path have been of continuing interest [74,75].
2. (WoW2 #329) For any graph G, G is König–Egerváry if and only if $\alpha(G) = \alpha_c(G)$, where α is the independence number of the graph, and α_c is the critical independence number of the graph. A graph is König–Egerváry if the independence number of the graph plus the matching number of the graph equals the number of vertices of the graph. These graphs are a generalization of bipartite graphs (the König–Egerváry Theorem guarantees that bipartite graphs have this property [58]). It has been of continuing interest to find new characterizations for this class of graphs [76–79]. Sophie's conjecture was proved in [64].

We conclude this section by emphasizing that the Dalmatian heuristic can be applied to the production of non-mathematical conjectures: there is no requirement that the object-types be mathematical. They can just as well be physical. The only requirement is that the object-types have real number invariants. Application of the Dalmatian heuristic in a given situation requires that the problem can be represented as a question about the value of a numerical invariant of an "object". The answer will be found in terms of relations of other numerical invariants of the object. So once the object, main invariant, and other invariants are specified, and one or more data examples are provided, conjectured bounds for the main invariant can be generated; and, assuming we have the values of the other invariants, these conjectures can be used to make predictions about the value of the main invariant. In a sequel to this paper we plan to discuss our ideas for using automated conjecture-making programs in the design of conscious robots.

We believe that automated conjecture-making of invariant and property relations have even broader applications than we have described here, and that researchers will experiment with conjecture-making programs in a variety of settings.

11. Discussion

Finally we discuss some observations from our use of the program.

1. *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. Graffiti's post-Dalmatian conjectures in geometry, chemistry, and graph theory provided evidence of domain-independence; we have provided further evidence. It has been claimed that different domains require different heuristics. 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 "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*, the one 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.

2. *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. Even the famous example of Fleming's discovery of penicillin is no counterexample: Fleming had been investigating anti-bacterial agents.

The success of mathematical discovery programs that have contributed to mathematical discovery can largely be explained by the fact that they were *designed* to advance specific mathematical problems. Fajtlowicz's Graffiti, DeLaVina's Graffiti.pc and the program Conjecturing discussed here contribute to advancing research on existing mathematical problems insofar as they produce bounds for mathematical invariants where bounds are of existing interest. Borwein and Bailey's programs contribute to advancing existing mathematical problems insofar as they produce closed form expressions for hypergeometric series. AGX contributes to advancing existing mathematical problems insofar as it finds characterizations of families of extremal graphs, McCune's theorem prover contributes to advancing research on existing mathematical problems insofar as it proved open conjectures. Part of the reason that these programs contribute to scientific discovery is that they were *designed* to contribute to scientific discovery, *designed* to do things of existing interest to mathematicians. Many mathematical discovery programs that were less successful than these were not designed to address specific problems of interest.

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. Discovery of an efficient algorithm for computing the independence number of a graph, for instance, would have explosive theoretical and practical consequences (as it would yield an efficient algorithm for every other NP-hard invariant). The discovery of an odd perfect number—whose existence has been an open question since at least the time of Euclid—would have far fewer mathematical consequences, and maybe no practical consequences.

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. In contrast, many mathematical papers do not explicitly or implicitly address any existing mathematical problem; the contribution, if any, to scientific discovery is considerably more tenuous. Some address a curious observation—perhaps that all small integers can be written as a sum of two primes—and attempt to explain that observation.

Some results in the mathematical literature are recorded in textbooks, passed down, reproved, extended, and generalized; others are never used—and forgotten. The utility of mathematics in the natural sciences and computer science is its raison d'etre, and primary source of value to society. Of course, mathematicians like Hardy have famously proclaimed their own desire to pursue mathematical truths that are completely without practical utility [80]. He specifically mentions that he did not want his research to be of any use in weapons design; it is worth noting that Hardy did not claim that he hoped his research was without mathematical utility—of course he was interested in advancing existing mathematical problems. The two examples he gives of "serious" but "useless" mathematics are the theorems that there are infinitely many prime numbers, and that the square root of two is irrational. But this mathematics addressed existing mathematical questions and was, thus, of (mathematical) utility: mathematicians were already interested in prime numbers, and they had speculated that all lengths are *commensurable* (the proof that $\sqrt{2}$ is irrational shows that they are not).

A researcher may attempt to design a conjecture-making program to produce statements that are "interesting" or "surprising" [81]—but if the produced statements do not advance any existing problem then they cannot be expected to make a scientific contribution regardless of how interesting or surprising they are. Colton, for instance, set his program HR to find relations between the integer invariants `sigma` (the sum of the divisors), `tau` (the number of divisors), and the integer property `isprime`. Among the conjectures his program produced was:

```
for all a (isprime (sigma(a)) ⇒ isprime(tau(a)))
```

Unless there is existing interest in necessary conditions for the primality of the sum of the divisors of an integer, or sufficient conditions for the primality of the number of divisors of an integer, this statement will not advance existing mathematical research.

Some limitations on the statements produced by a conjecture-making program may be required in order for the produced statements to be of use to researchers—they may, for instance, need to be relatively simple in form. Human researchers often do not have any feeling for the meaning, much less the truth, of very complicated statements. What Hardy calls the "depth" of a statement could partly be measured by its (syntactic) complexity. It is often beyond human mathematicians abilities to wrestle with complex statements involving multiple invariants. Computers may be able to investigate deeper statements than humans are able to. As automated theorem-proving programs improve, complicated conjectures would be of use—they could be proved, and applied, by machines; complexity aside, true statements can be used in making predictions and guiding behavior.

3. *Intelligent conjecture-making programs do not require sophisticated data-mining techniques.* An interesting feature of programs implementing the Dalmatian heuristic is that they can make conjectures based on only a few examples. In this way they are similar to human conjecture-makers. If added examples are counterexamples to previously made conjectures, then each of the examples is "significant" to the program: no useless examples are stored or required. Again this is similar to what humans do. We don't remember every example—only significant ones.

4. *Developers of programs that contribute to scientific discovery should not try to simulate human scientific discovery.* If you could design a program that does whatever it is that humans do when they make scientific discoveries then, of course, you have solved the problem of automating scientific discovery. But a program may be counted as intelligent while doing things in distinctly non-human ways. The chess computer Deep Blue can certainly be said to play intelligent chess. Since the success of the program is, at least in part, due to its ability to evaluate hundreds of millions of positions per second—far beyond human capabilities—Deep Blue is not said to simulate human chess playing.

Some researchers have aimed to simulate human scientific discovery [82]. Others have aimed to produce programs that contributed to new scientific discoveries. Some hoped to do both. These differences and tensions date to the earliest days of this research. Wang's criticism of Simon and Newell's Logic Theorist was that their program wasn't actually very good at proving theorems. Wang produced a program that did a much better job at proving the same body of test theorems [6]. Simon and Newell's response was that their goal was *not* to develop a program whose primary aim was to prove theorems but, rather, to develop a program that proves theorems *in the way that* humans do, to "simulate" human theorem-proving abilities. Wang made no claim that his program simulated human theorem proving.

Lenat's work on AM is often cited as a contribution to research on the automation of mathematical discovery [81]. But AM was not designed to make mathematical discoveries: that is, it was not designed to produce statements that addressed any existing mathematical problems. And it did not make any mathematical discoveries. It was designed to do something else entirely—to simulate human mathematical discovery—and should be evaluated by the standards of simulation research. Research on automated scientific discovery must look to, and build on, the successful ideas in discovery research. Confusion on the issue of simulation versus discovery dates to the earliest days of this research. Wang reports that he read Lenat's dissertation (about the development of his AM program), and writes that he "could not see how one might further build on such a baffling foundation" [83]. What Wang failed to understand is that Lenat's program was a contribution to simulation research and not designed to contribute to mathematical research. And the confusion continues to the present day—which, if not corrected, can hinder research on programs meant to make contributions to mathematical discovery.

Acknowledgements

Appendix A. Acquiring and using the program

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 [84]. Sage is intended as a free replacement for general mathematical software programs such as Maple, Matlab and Mathematica. In contrast with proprietary programs, anyone can examine, correct, and improve Sage's included algorithms and code. Sage users who develop programs for their own research are encouraged to include them in the public distribution of Sage, for general use; the user base, and community of developers, are large and growing. Versions of Sage for all major operating systems, manuals, and documentation are available at: www.sagemath.org. Examples, and the use of our program, are discussed below.

Our program is designed as a Sage package. The attraction of Sage for this project stems from the fact that it is free, that it is easy-to-use, that it has a large number of built-in invariants for a variety of mathematical objects, and that other researchers can easily use our code. Sage uses Python as its interface language and includes well-known packages such as GAP for computer algebra, SINGULAR for algebraic geometry, PARI for number theory, LAPACK for linear algebra, R for statistical computation, NumPy and SciPy for numerical computing, and CVXOPT for convex optimization, linear and semidefinite programming.

The first part of the program is the expression generator described earlier. This C program is wrapped into a Sage package which allows it to be installed with one command. The second part of the program, containing the main loop, is the Python file `conjecturing.py`. This file can be loaded into Sage to provide a seamless integration of the expressions generator using the Dalmatian heuristic. It provides a method `conjecture` that—in its most basic form—takes three arguments: a list of objects, a list of invariants and a main invariant, that is, the invariant for which one wants to find a bound. The objects can be any (Sage) objects; the only provision is that the invariants are able to translate the objects to numerical values. The invariants must be functions of a single argument (the inputs are the objects). The main invariant is represented by the index of its location in the list of invariants.

The conjecture-making code was partly based on Patrick Gaskill's code for a related project, and is available at: github.com/IndependenceNumberProject/inp.

What follows is a short example of how the `conjecture` method can be used in Sage.

```
sage: def max_degree(g):
....:     return max(g.degree())
....:
sage: invariants = [Graph.size, Graph.order, max_degree]
sage: objects = [graphs.CompleteGraph(n) for n in [3,4,5]]
sage: conjecture(objects, invariants, 0)
[size(x) <= 2*order(x),
 size(x) <= max_degree(x)^2 - 1,
 size(x) <= 1/2*order(x)*max_degree(x)]
```

The first two lines of this example define a new (that is, not existing in Sage) graph theoretic invariant `max_degree`. The next line specifies `invariants` to be a list of built-in and user-defined invariants; these will potentially appear in the conjectured expressions; and the user can remove any of these and add as many others as she likes. The fourth line specifies `objects` to be a list of graphs that will be used by the program when making conjectures—in this case the list is initialized with three complete graphs. Again, the user can remove any of these and add arbitrarily many others. The `conjecture` function takes the lists of invariants and objects, together with a number specifying which of the invariants in the `invariants` list should be used as the main invariant; in this case, 0 is input, indicating that `invariants[0]`—namely, `Graph.size`—should be used as the main invariant. The last three lines are the output of the program, three conjectured upper bounds for the size of a graph. All three conjectures happen to be false: they are all necessarily true for the complete graphs on 3, 4 and 5 vertices—but are false for other graphs not in the input list.

Scripts that generate the conjectures for most of the reported runs of our program are also available at: nvcleemp.github. io/conjecturing/. The scripts used to generate the conjectures above are listed in Table 18. The purpose of providing these scripts is two-fold. They allow for the reproducibility of our results, and they also provide models for researchers to imitate in generating their own conjectures. Exact reproducibility will depend on whether the researcher is using a machine that is the same speed as the second author's. The expression generation program times out after 5 seconds. A user on a faster machine will generate more expressions to test for truth and significance than our machine did; one on a slower machine will generate fewer expressions. In either case, there is then a possibility of ending up with slightly different conjectures than the conjectures we report below.

All invariants used in our experiments are either built-in Sage functions or user-defined procedures. The code for the user-defined procedures can be found in the files: `matrixtheory.py`, `numbertheory.py`, and `graphtheory.py`.

Here are more details about the generation of the matrix theory conjectures. The first rounds of conjectures used symmetric 2×2 matrices with integer entries ranging from -10 to 10. This was to allow systematic generation of a family of examples. In fact, 2×2 matrices turned out to be too special of a subclass of matrices to lead to good conjectures—other researchers quickly found 3×3 matrices that were counterexamples to many of these.

Furthermore, the specialness of these examples led to non-general conjectures. In one run, for instance, the program conjectured that, for any symmetric matrix, the absolute value of the determinant is at least as large as the smallest eigenvalue. In fact, this conjecture *is* true for 2×2 matrices with integer entries. But it is certainly not true in general (a diagonal matrix with $\frac{1}{2}$'s on the diagonal is a counterexample).

Table 18
Associated scripts for conjecture runs.

Conjectures	Script
Table 2	`determinant_upper_bound_conjectures.py`
Table 3	`determinant_lower_bound_conjectures.py`
Table 4	`abs_determinant_upper_bound_conjectures1.py`
Table 5	`abs_determinant_lower_bound_conjectures2.py`
Table 6	`goldbach_conjectures1.py`
Table 7	`goldbach_conjectures2.py`
Table 8	`prime_pi_conjectures1.py`
Table 10	`riemann_conjectures.py`
Table 9	`riemann_conjectures.py`
Table 11	`domination_upper_bound_conjectures.py`
Table 13	`domination_upper_bound_conjectures2.py`
Table 14	`domination_lower_bound_conjectures.py`

We expect that generation of larger matrices with a larger range of entries would lead to better conjectures. In this case it will become impossible to systematically generate any interesting class, and a researcher may experiment with choosing randomly generated matrices from a chosen class.

McKay's program *geng*, which we used for generating all graphs up to any specified order, is freely available for non-commercial use and works easily with Sage.

References

[1] S. Fajtlowicz, On conjectures of Graffiti. V, in: Graph Theory, Combinatorics, and Algorithms, vols. 1, 2, Kalamazoo, MI, 1992, in: Wiley-Intersci. Publ., Wiley, New York, 1995, pp. 367–376.
[2] E. DeLaVina, Graffiti.pc: a variant of Graffiti, DIMACS Ser. Discret. Math. Theor. Comput. Sci. 69 (2005) 71.
[3] A. Turing, Intelligent machinery, in: The Essential Turing, 2004, pp. 395–432.
[4] H.A. Simon, A. Newell, Heuristic problem solving: the next advance in operations research, Oper. Res. 6 (1) (1958) 1–10.
[5] W. McCune, Solution of the Robbins problem, J. Autom. Reason. 19 (3) (1997) 263–276.
[6] H. Wang, Toward mechanical mathematics, IBM J. Res. Dev. 4 (1960) 2–22.
[7] S. Fajtlowicz, On conjectures of Graffiti, in: Proceedings of the First Japan Conference on Graph Theory and Applications, Hakone, 1986, vol. 72, 1988, pp. 113–118.
[8] S. Fajtlowicz, On conjectures of Graffiti. III, in: Nineteenth Southeastern Conference on Combinatorics, Graph Theory and Computing, Baton Rouge, LA, 1988, in: Congr. Numer., vol. 66, 1988, pp. 23–32.
[9] S. Fajtlowicz, On conjectures of Graffiti. II, in: Eighteenth Southeastern International Conference on Combinatorics, Graph Theory and Computing, Boca Raton, FL, 1987, in: Congr. Numer., vol. 60, 1987, pp. 189–197.
[10] S. Fajtlowicz, On conjectures of Graffiti. IV, in: Proceedings of the Twentieth Southeastern Conference on Combinatorics, Graph Theory, and Computing, Boca Raton, FL, 1989, vol. 70, 1990, pp. 231–240.
[11] B.A. Cipra, The sorcerer's apprentice, Science 244 (4906) (1989) 770.
[12] E. DeLaVina, Some history of the development of Graffiti, in: Graphs and Discovery, in: DIMACS Ser. Discret. Math. Theor. Comput. Sci., vol. 69, Amer. Math. Soc., Providence, RI, 2005, pp. 81–118.
[13] C.E. Larson, A survey of research in automated mathematical conjecture-making, in: Graphs and Discovery, in: DIMACS Ser. Discret. Math. Theor. Comput. Sci., vol. 69, Amer. Math. Soc., Providence, RI, 2005, pp. 297–318.
[14] D.B. Lenat, The ubiquity of discovery, Artif. Intell. 9 (3) (1977) 257–285.
[15] D.B. Lenat, On automated scientific theory formation: a case study using the am program, Mach. Intell. 9 (1979) 251–286.
[16] D.B. Lenat, The nature of heuristics, Artif. Intell. 19 (2) (1982) 189–249.
[17] R. Davis, D.B. Lenat, Knowledge-Based Systems in Artificial Intelligence, McGraw-Hill International Book Co., 1982.
[18] S.L. Epstein, On the discovery of mathematical theorems, in: IJCAI, 1987, pp. 194–197.
[19] S.L. Epstein, Learning and discovery: one system's search for mathematical knowledge, Comput. Intell. 4 (1) (1988) 42–53.
[20] S. Colton, A. Bundy, T. Walsh, Automatic concept formation in pure mathematics, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence, IJCAI'99, vol. 2, Morgan Kaufmann Publishers, 1999, pp. 786–791.
[21] S. Colton, Refactorable numbers—a machine invention, J. Integer Seq. 2 (99.1) (1999) 2.
[22] S. Colton, Automated Theory Formation in Pure Mathematics, Springer, Heidelberg, 2002.
[23] S. Colton, Automated conjecture making in number theory using HR, Otter and Maple, J. Symb. Comput. 39 (5) (2005) 593–615.
[24] G. Caporossi, P. Hansen, Variable neighborhood search for extremal graphs: 1 the autographix system, Discrete Math. 212 (1) (2000) 29–44.
[25] G. Caporossi, P. Hansen, Variable neighborhood search for extremal graphs. 5. Three ways to automate finding conjectures, Discrete Math. 276 (1) (2004) 81–94.
[26] M. Aouchiche, G. Caporossi, P. Hansen, M. Laffay, Autographix: a survey, Electron. Notes Discrete Math. 22 (2005) 515–520.
[27] H. Mélot, Facet defining inequalities among graph invariants: the system graphedron, Discrete Appl. Math. 156 (10) (2008) 1875–1891.
[28] J. Christophe, S. Dewez, J.-P. Doignon, G. Fasbender, P. Grégoire, D. Huygens, M. Labbé, S. Elloumi, H. Mélot, H. Yaman, Linear inequalities among graph invariants: using graphedron to uncover optimal relationships, Networks 52 (4) (2008) 287–298.
[29] D. Cvetković, I. Gutman, The computer system GRAPH: a useful tool in chemical graph theory, J. Comput. Chem. 7 (5) (1986) 640–644.
[30] R. Brigham, R. Dutton, INGRID: a software tool for extremal graph theory research, Congr. Numer. 39 (1983) 337–352.
[31] R.D. Dutton, R.C. Brigham, F. Gomez, INGRID: a graph invariant manipulator, J. Symb. Comput. 7 (2) (1989) 163–177.
[32] R. Bagai, V. Shanbhogue, J.M. Żytkow, S.-C. Chou, Automatic theorem generation in plane geometry, in: Methodologies for Intelligent Systems, Springer, 1993, pp. 415–424.
[33] R. Bagai, V. Shanbhogue, J.M. Zytkow, S.-C. Chou, Discovery of geometry theorems: avoiding isomorphic situation descriptions, in: Proceedings of the Fifth International Conference on Computing and Information, ICCI'93, IEEE, 1993, pp. 354–358.
[34] H.S. Wilf, D. Zeilberger, Towards computerized proofs of identities, Bull. Am. Math. Soc. 23 (1) (1990) 77–83.
[35] J.M. Borwein, D.H. Bailey, Mathematics by Experiment: Plausible Reasoning in the 21st Century, AK Peters, Natick, MA, 2004.

C.E. Larson, N. Van Cleemput / Artificial Intelligence 231 (2016) 17–38

[36] F.R.K. Chung, The average distance and the independence number, J. Graph Theory 12 (2) (1988) 229–235.
[37] P. Erdős, M. Saks, V.T. Sós, Maximum induced trees in graphs, J. Comb. Theory, Ser. B 41 (1) (1986) 61–79.
[38] S. Fajtlowicz, A characterization of radius-critical graphs, J. Graph Theory 12 (4) (1988) 529–532.
[39] O. Favaron, M. Mahéo, J.-F. Saclé, On the residue of a graph, J. Graph Theory 15 (1) (1991) 39–64.
[40] Jerrold R. Griggs, Daniel J. Kleitman, Independence and the Havel–Hakimi residue, Discrete Math. 127 (1–3) (1994) 209–212, Graph theory and applications (Hakone, 1990).
[41] P. Hansen, A. Hertz, R. Kilani, O. Marcotte, D. Schindl, Average distance and maximum induced forest, J. Graph Theory 60 (1) (2009) 31–54.
[42] D. Logothetti, H.S.M. Coxeter, An interview with H.S.M. Coxeter, the king of geometry, Two-Year Coll. Math. J. 11 (1) (1980) 2–19.
[43] P.W. Fowler, D.E. Manolopoulos, An Atlas of Fullerenes, Clarendon Press, Oxford, 1995.
[44] P.W. Fowler, K.M. Rogers, S. Fajtlowicz, P. Hansen, G. Caporossi, Facts and conjectures about fullerene graphs: leapfrog, cylinder and Ramanujan fullerenes, in: Algebraic Combinatorics and Applications, Gößweinstein, 1999, Springer, Berlin, 2001, pp. 134–146.
[45] S. Fajtlowicz, C.E. Larson, Graph-theoretic independence as a predictor of fullerene stability, Chem. Phys. Lett. 377 (5–6) (2003) 485–490.
[46] D. Stevanović, G. Caporossi, On the (1, 2)-spectral spread of fullerenes, in: Graphs and Discovery, in: DIMACS Ser. Discret. Math. Theor. Comput. Sci., vol. 69, Amer. Math. Soc., Providence, RI, 2005, pp. 365–370.
[47] S.M. Daugherty, Independent sets and closed-shell independent sets of fullerenes, PhD thesis, Victoria, B.C., Canada, 2010, AAINR66829.
[48] R.B. Mallion, D.H. Rouvray, The golden jubilee of the Coulson–Rushbrooke pairing theorem, J. Math. Chem. 5 (1) (1990) 1–21.
[49] A. Peeters, GrInvIn – a software framework for education and research in graph theory, Thesis (Ph.D.), Ghent University, 2008.
[50] A. Peeters, K. Coolsaet, G. Brinkmann, N. Van Cleemput, V. Fack, GrInvIn in a nutshell, J. Math. Chem. 45 (2) (2009) 471–477.
[51] R.A. Horn, C.R. Johnson, Matrix Analysis, 2nd edition, Cambridge University Press, 2012.
[52] G.H. Hardy, E.M. Wright, An Introduction to the Theory of Numbers, 5th edition, Oxford University Press, 1979.
[53] J. Richstein, Verifying the Goldbach conjecture up to $4 \cdot 10^{14}$, Math. Comput. 70 (236) (2001) 1745–1749.
[54] R.E. Crandall, C. Pomerance, Prime Numbers: A Computational Perspective, vol. 182, Springer, 2005.
[55] B. Rosser, Explicit bounds for some functions of prime numbers, Am. J. Math. 63 (1) (1941) 211–232.
[56] J.B. Rosser, L. Schoenfeld, Approximate formulas for some functions of prime numbers, Ill. J. Math. 6 (1962) 6–9.
[57] T.W. Haynes, S.T. Hedetniemi, P.J. Slater, Fundamentals of Domination in Graphs, Monogr. Textb. Pure Appl. Math., vol. 208, Marcel Dekker Inc., New York, 1998.
[58] D.B. West, Introduction to Graph Theory, 2nd edition, Prentice Hall, 2001.
[59] R. Pepper, On the annihilation number of a graph, Recent Adv. Appl. Math. Comput. Inf. Sci. 1 (2009) 217–220.
[60] C.E. Larson, R. Pepper, Graphs with equal independence and annihilation numbers, Electron. J. Comb. 18 (1) (2011).
[61] B.D. McKay, Nauty users guide (version 2.4), Computer Science Dept., Australian National University, 2007.
[62] Michael R. Garey, David S. Johnson, Computers and intractability, in: A Guide to the Theory of NP-Completeness, in: Ser. Books Math. Sci., W.H. Freeman and Co., San Francisco, Calif., 1979.
[63] C.E. Larson, A note on critical independence reductions, Bull. Inst. Comb. Appl. 51 (2007) 34–46.
[64] C.E. Larson, The critical independence number and an independence decomposition, Eur. J. Comb. 32 (2) (2011) 294–300.
[65] E. DeLaVina, C.E. Larson, A parallel algorithm for computing the critical independence number and related sets, Ars Math. Contemp. (2015), in press.
[66] E. Angel, R. Campigotto, C. Laforest, A new lower bound on the independence number of graphs, Discrete Appl. Math. 161 (6) (2013) 847–852.
[67] L. Lovász, On the shannon capacity of a graph, IEEE Trans. Inf. Theory 25 (1) (1979) 1–7.
[68] Donald E. Knuth, The sandwich theorem, Electron. J. Comb. 1 (1994) 1, 48 pp. (electronic).
[69] L. Lovász, Semidefinite programs and combinatorial optimization, in: Recent Advances in Algorithms and Combinatorics, in: CMS Books Math./Ouvrages Math. SMC, vol. 11, Springer, New York, 2003, pp. 137–194.
[70] R.C. Read, R.J. Wilson, An Atlas of Graphs, vol. 21, Clarendon Press, Oxford, 1998.
[71] Béla Bollobás, Random Graphs, second edition, Cambridge Stud. Adv. Math., vol. 73, Cambridge University Press, Cambridge, 2001.
[72] D. Gale, A curious Nim-type game, Am. Math. Mon. (1974) 876–879.
[73] E. DeLaVina, R. Pepper, W. Waller, Independence, radius and Hamiltonian paths, MATCH Commun. Math. Comput. Chem. 58 (2) (2007) 481–510.
[74] R.J. Gould, Updating the Hamiltonian problem—a survey, J. Graph Theory 15 (2) (1991) 121–157.
[75] R.J. Gould, Advances on the Hamiltonian problem—a survey, Graphs Comb. 19 (1) (2003) 7–52.
[76] R.W. Deming, Independence numbers of graphs—an extension of the Koenig–Egervary theorem, Discrete Math. 27 (1) (1979) 23–33.
[77] F. Sterboul, A characterization of the graphs in which the transversal number equals the matching number, J. Comb. Theory, Ser. B 27 (2) (1979) 228–229.
[78] L. Lovász, Ear-decompositions of matching-covered graphs, Combinatorica 3 (1) (1983) 105–117.
[79] J.-M. Bourjolly, W.R. Pulleyblank, König–Egerváry graphs, 2-bicritical graphs and fractional matchings, Discrete Appl. Math. 24 (1) (1989) 63–82.
[80] G.H. Hardy, A Mathematician's Apology, The University Press, 1940.
[81] S. Colton, A. Bundy, T. Walsh, On the notion of interestingness in automated mathematical discovery, Int. J. Hum.-Comput. Stud. 53 (3) (2000) 351–375.
[82] P. Langely, H.A. Simon, G.L. Bradshaw, J.M. Zytkow, Scientific Discovery: Computational Explorations of the Creative Process, MIT Press, Cambridge, MA, 1987.
[83] H. Wang, Computer theorem proving and artificial intelligence, in: Computation, Logic, Philosophy, Springer, 1990, pp. 63–75.
[84] W. Stein, Sage: open source mathematical software (version 2.10.2), The Sage group, http://www.sagemath.org, 2008.