Machine Theorem Discovery

Fangzhen Lin

■ *This article describes a framework for machine theorem discovery and illustrates its use in discovering state invariants in planning domains and properties about Nash equilibria in game theory. It also discusses its potential use in program verification in software engineering. The main message of the article is that many AI problems can and should be formulated as machine theorem discovery tasks.*

The idea of using a computer program to help discover theorems is an old one and will not surprise anyone nowadays. In late 1950s, logician Hao Wang at IBM wrote a code called Program II (Wang 1960) to discover deep theorems in propositional logic. For Wang, a theorem is deep if it is short but requires a long proof. Understandably, Wang's program did not produce any really deep theorems, perhaps for the simple reason that there are no tautologies that are short and require a long proof.

A better-known example in AI is the Automated Mathematician (AM) that Douglas Lenat developed in 1976 as part of his PhD dissertation (Lenat 1979). While not strictly about discovering theorems, AM uses heuristic search to simulate how mathematicians discover interesting concepts and conjectures in number theory. It caused some excitement in the community as Lenet claimed that one of the Lisp functions it generated represented the fundamental theorem of arithmetic.

Input

1 Language description: sorts, predicates, constants, functions.

2 A collection of sets of objects to construct models of the language.

3 A model confirmation relation between models M and sentences φ: M confirms φ if M satisfies the required property about φ, provided that M is a model of the domain theory.

4 (Optional) A theorem prover for checking if a sentence is a theorem in the given theory.

5 A specification of the space of hypotheses: by defaults, all atomic sentences, binary clauses, type information (unary predicates), functional dependencies will be in the space. The user can also specify more.

Output: Weakest conjectures.

Figure 1. An Algorithm for Machine Theorem Discovery.

A more impressive system is Siemion Fajtlowicz's Graffiti (Fajtlowicz 1988), which generates conjectures in graph theory. More recently, Craig Larson extended Fajtlowicz's Graffiti into a more general framework for mathematical conjecture making in discrete math (Larson and Cleemput 2016; 2017). For an early survey on these systems, see Larson (2005).

However, these examples all focus on a general theory like propositional logic, number theory, or graph theory. For AI applications, more often we want to prove theorems in a very specific theory for some very specific purpose. For instance, we could be given a planning domain such as the logistics domain, and we want to know if there are special properties about the domain that can help prune the search space for the planner in hand. Finding these properties is the same as discovering theorems, once the domain is axiomatized by a formal theory, for example, in the situation calculus. As another example, in software verification, we may want to prove that a certain postcondition holds when the input satisfies a certain precondition. This relationship can be naturally formulated as a theorem-proving problem and is, in general, intractable. It helps, then, to discover some useful properties about the program in question to make the theorem-proving task easier. In fact, much of the formal work on verifying sequential programs can be said to be about discovering suitable loop invariants as pioneered in Hoare's logic (Hoare 1969).

In the past, my colleagues and I have developed programs for discovering state invariants in planning (Lin 2004), for discovering theorems that capture strong equivalence in logic programming (Lin and Chen 2007), for identifying conditions that capture the uniqueness of pure Nash equilibria in games in normal form (Tang and Lin 2011a), and for checking the consistency of social choice axioms by translating them to SAT (Tang and Lin 2009). Our work has inspired others to apply similar approaches to other problems (for example, Geist and Endriss 2011; Brandl et al. 2015; Brandt and Geist 2016; Brandl, Brandt, and Geist 2016). For instance, instead of SAT, Brandl, Brandt, and Geist (2016) used SMT in their work on showing the incompatibility of certain notions of efficiency and strategy-proofness.

In this article, I will outline a general approach to machine theorem discovery. I will review some of our earlier work and recast it in a general framework. As it will become clear, the key is to formulate hypotheses in a formal logical language.

Theorem Discovery

In a nutshell, theorem discovery can be viewed as an iterative two-step process:

while (true) do

1. Find a reasonable conjecture;

2. Verify the conjecture.

Of course, it all comes down to what reasonable conjectures are, how to search for them, and once they've been found, how to verify them as a theorem of the given background theory. A key point of this article is that current knowledge representation (KR) formalisms, particularly those of first-order logic, are good for specifying the space of conjectures. Furthermore, solvers such as those for SAT and CSP go a long way in pruning the search space, identifying good conjectures, and verifying them in the end.

In this article I focus on machine theorem discovery using the algorithm depicted in figure 1. Here, a conjecture is a hypothesis that is confirmed by all models that can be constructed using the domains given in input 2, and a conjecture is a weakest one if there is no other conjecture that is properly entailed by it. If the optional theorem prover in input 4 is given, it is used to check entailment. Otherwise, entailment is decided using all models that can be constructed using the domains given in input 2. If no theorem prover is given, then the weakest conjectures thus generated may be too strong, and the system may miss some possible conjectures. For example, if conjecture A implies another conjecture B in all constructed models, A will be pruned. However, it may well be possible that A does not entail B in general.

I will illustrate this algorithm by showing how two of our earlier systems — one for discovering state invariants in planning, and the other on discovering theorems about pure Nash equilibria — can be cast in this general framework.

Discovering State Invariants in Dynamic Systems

The state of a dynamic system can change according to some natural events or as a result of some actions done by agents. However, there are some properties — for example, that at any given time an object can be at only one location — that will persist no matter what. These properties are called *state invariants* and they are useful in planning for pruning search spaces. In fact, several systems have been designed specifically for learning state constraints (Huang, Selman, and Kautz 2000; Gerevini and Schubert 1998).

To make learning state invariants a theorem discovery problem, we need to make precise the notion of state invariants, as well as to formalize the given planning domain.

A dynamic system consists of states and the transitions between states. A planning domain is like a dynamic system, but typically with states described by a logic language and transitions by actions. In my paper on state invariants (Lin 2004), I start with a domain language to describe states and extend it with actions. Briefly, states are models of a first-order language called a *domain language*. As usual, propositions in the domain language whose truth values can be changed by actions are called *fluents*. To formalize the effects of actions, we extend the domain language with a special sort action. For each predicate in the domain language, we introduce a new predicate with the same name but with an extra argument of sort action. Intuitively, if p is a fluent, and a an action, then $p(a)$ denotes the truth value of p after a is successfully performed in the current state. I call these new predicates *successor state predicates*. For instance, if *clear* is a fluent, then we write *clear*(x) to mean that block x is clear in the given initial state and we write *clear*(x, *unstack*(x, y)) to mean that x is clear in the successor situation of doing *unstack*(x, y) in the initial state. I also assume a special unary predicate, *Poss*, to denote the action precondition. Thus, *Poss*(*unstack*(x, y)) will stand for the precondition of doing *unstack*(x, y).

So formally, an action theory is a family of first-order theories $\{T_A \mid A \text{ is an action type}\}$, where for each action type A, T_A consists of the following axioms:

An action precondition axiom of the form

$$\forall \mathbf{x}.Poss(A(\mathbf{x})) \equiv \Psi, \tag{1}$$

where Ψ is a formula in the domain language whose free variables are in $\mathbf{x}$. (Thus Ψ cannot mention *Poss* and any successor state predicates.)

For each domain predicate F, an axiom of the form

$$(\forall \mathbf{x}, \mathbf{y}).F(\mathbf{x}, A(\mathbf{y})) \equiv \Phi_F(\mathbf{x}, \mathbf{y}), \tag{2}$$

where $\mathbf{x}$ and $\mathbf{y}$ do not share common variables, and Φ_F is a formula in the domain language whose free variables are from $\mathbf{x}$ and $\mathbf{y}$.

Example 1

In the blocks world, for the action type *stack*, we have the following axioms (all free variables below are universally quantified from outside):

$Poss(stack(x, y)) \equiv holding(x) \wedge clear(y)$,
$on(x, y, stack(u, v)) \equiv (x = u \wedge y = v) \vee on(x, y)$,
$ontable(x, stack(u, v)) \equiv ontable(x)$,
$handempty(stack(u, v)) \equiv true$,
$holding(x, stack(u, v)) \equiv false$,
$clear(x, stack(u, v)) \equiv clear(x) \wedge x \neq v$.

The following definition captures the intuition that a state invariant is a formula that if true initially, will continue to be true after the successful completion of every possible action.

Definition 1

Given an action theory $\{T_A \mid A \text{ is an action type}\}$, a formula W in the domain language is a state invariant if for each action type A,

$$T_A \models \forall \mathbf{y}.W \wedge Poss(A(\mathbf{y})) \supset W(A(\mathbf{y})), \tag{3}$$

where $W(A(\mathbf{y}))$ is the result of replacing each atom $F(\mathbf{t})$ in W by $F(\mathbf{t}, A(\mathbf{y}))$, and $\models$ is the logical entailment in first-order logic.

Thus, the problem of discovering and learning state invariants becomes a problem of discovering theorems of the form (equation 3). Based on this set-

up, I described a system (Lin 2004) for discovering various types of state invariants in planning domains such as functional constraints, type constraints, and domain closure constraints. The system performs remarkably well for almost all known benchmark planning domains. For instance, for the popular logistics domain, it returns a complete set of state invariants in the sense that a state is "legal" if it satisfies all the constraints.

In terms of the general algorithm outlined in the last section, to discover state invariants in a planning domain, the required inputs are as follows:

Language: domain language extended with actions.

Sets of objects: for each sort, sets of up to a small number, say three, objects.

Model confirmation: given an interpretation M for the domain language, and a sentence φ in domain language, M confirms φ if for every action A, when M is extended to be a model of T_A, it satisfies $\forall \mathbf{y}.\varphi \wedge Poss(A(\mathbf{y})) \supset \varphi(A(\mathbf{y}))$. In other words, φ is a state invariant in M according to equation 3.

Space of hypotheses:

Functional constraints: sentences of the form

$$\forall x, y_1, y_2.at(x, y_1) \wedge at(x, y_2) \supset y_1 = y_2.$$

Type constraints: sentences of the form

$$\forall x, y.at(x, y) \supset [package(x) \vee vehicle(x)] \wedge location(y).$$

Domain closure constraints: these constraints say that certain objects must belong to one of the predefined categories, for example, a package needs to be either at a location or inside a vehicle.

A precise definition of these constraints can be found in Lin (2004).

A theorem in Lin (2004) shows that a theorem prover is often not needed, as it can be replaced by model checking. For many action theories and for most of the constraints considered here, once a constraint is confirmed by all models up to a certain finite size, it is guaranteed to be a state invariant. A similar, but simpler theorem is given in the next section in the context of discovering theorems in game theory.

Two-Person Games

As another example of machine theorem discovery, consider the problem of finding conditions under which a two-person game has a unique pure Nash equilibria payoff. The key idea again is to formulate the background theory and the hypotheses in first-order logic. This was the joint work I undertook with Pingzhong Tang (Tang and Lin 2011a).

A two-person game is a tuple $(A, B, \leq_1, \leq_2)$, where A and B are sets of (pure) strategies of players 1 and 2, respectively, and $\leq_1$ and $\leq_2$ are total orders on $A \times B$, called *preference relations* for players 1 and 2, respectively.

For each $b \in B$, the set of best responses by player 1 to the action b by player 2 is defined as follows:

$$B_1(b) = \{a \mid a \in A, \text{ and for all } a' \in A, (a', b) \leq_1 (a, b)\}.$$

Similarly, for each $a \in A$, the set of best responses by player 2 is:

$$B_2(a) = \{b \mid b \in B, \text{ and for all } b' \in B, (a, b') \leq_2 (a, b)\}$$

A profile $(a, b) \in A \times B$ is a (pure-strategy) Nash equilibrium if both $a \in B_1(b)$ and $b \in B_2(a)$. A game can have one, more than one, or no Nash equilibria. Properties of pure Nash equilibria have been studied extensively. For instance, if a game is strictly competitive (Friedman 1983; Moulin 1976), then it has a unique Nash equilibria payoff in the sense that if both s and s' are Nash equilibria of the game, then the payoffs for them are the same for each player, that is, $s \leq_i s'$ and $s' \leq_i s$ for $i = 1, 2$. This outcome is also true for weakly unilaterally competitive games (Kats and Thisse 1992). It is also known that ordinal potential games (Topkis 1998) and super-modular games (Monderer and Shapley 1996) always have Nash equilibria.

I now show how two-person games and some key notions about them can be formulated in first-order logic.

Consider a first-order language with two sorts, α and β. Sort α is for player 1's actions, and β for player 2's actions. In the following, I use variables $x, x_1, x_2, ..$ to range over α, and $y, y_1, y_2, ...$ to range over β. The players' preferences are represented by two infix predicates $\leq_1$ and $\leq_2$ of the type $(\alpha, \beta) \times (\alpha, \beta)$. Given that the preferences are total orders, these two predicates need to satisfy the following axioms (all free variables in a displayed formula are assumed to be universally quantified from outside):

$$(x, y) \leq_i (x, y), \tag{4}$$

$$(x_1, y_1) \leq_i (x_2, y_2) \vee (x_2, y_2) \leq_i (x_1, y_1), \tag{5}$$

$$(x_1, y_1) \leq_i (x_2, y_2) \wedge (x_2, y_2) \leq_i (x_3, y_3) \supset (x_1, y_1) \leq_i (x_3, y_3), \tag{6}$$

where $i = 1, 2$. In the following, I denote by Σ the set of these sentences. Thus, two-person games correspond to first-order models of Σ, and two-person finite games correspond to first-order finite models of Σ.

I write $(x_1, y_1) <_i (x_2, y_2)$ as shorthand for

$$(x_1, y_1) \leq_i (x_2, y_2) \wedge \neg(x_2, y_2) \leq_i (x_1, y_1),$$

and now the condition for a profile (ξ, ζ) to be a Nash equilibrium is captured by the following formula:

$$\forall x.(x, \zeta) \leq_1 (\xi, \zeta) \wedge \forall y.(\xi, y) \leq_2 (\xi, \zeta) \tag{7}$$

In the following, I shall denote the above formula by $NE(\xi, \zeta)$. The following sentence expresses the uniqueness of Nash equilibria payoff:

$$NE(x_1, y_1) \wedge NE(x_2, y_2) \supset (x_1, y_1) \simeq_1 (x_2, y_2) \wedge (x_1, y_1) \simeq_2 (x_2, y_2), \quad (8)$$

where for $i = 1, 2$, $(x_1, y_1) \simeq_i (x_2, y_2)$ is shorthand for

$$(x_1, y_1) \leq_i (x_2, y_2) \wedge (x_2, y_2) \leq_i (x_1, y_1).$$

A game is strictly competitive if it satisfies the following property:

$$(x_1, y_1) \leq_1 (x_2, y_2) \equiv (x_2, y_2) \leq_2 (x_1, y_1). \quad (9)$$

Thus, it should follow that

$$\Sigma \models (9) \supset (8). \quad (10)$$

Notice that I have assumed that all free variables in a displayed formula are universally quantified from outside. Thus, formula 9 is a sentence of the form $\forall x_1, x_2, y_1, y_2 \varphi$. Similarly for formula 8.

Thus, to discover conditions Q for a game to have unique Nash equilibria payoff is to discover theorems of the form $\Sigma \models Q \supset (8)$. Of course, there are an infinite number of such theorems, so practically speaking, we have to restrict the type of conditions we want a program to search for. A good starting point is to try prenex formulas Q of the following form:

$$\exists \mathbf{x}_1 \exists \mathbf{y}_1 \forall \mathbf{x}_2 \forall \mathbf{y}_2 Q_0$$

where Q_0 does not mention any quantifiers. A theorem from Lin (2007) says that such Q is a sufficient condition for the uniqueness of Nash equilibria iff it is so for all games of sizes up to $(|\mathbf{x}_1| + 2) \times (|\mathbf{y}_1| + 2)$, and it is a sufficient condition for the nonexistence of Nash equilibria iff it is so for all games of sizes up to $(|\mathbf{x}_1|+1) \times (|\mathbf{y}_1|+1)$. Effectively, this reduces theorem proving such as showing $\Sigma \models Q \supset (8)$ to finite model checking.

This also holds for many specialized games, such as the class of strict games: a game is strict if for both players, different profiles have different payoffs, that is, $(a, b) = (a', b')$ whenever $(a, b) \leq_i (a_1, b_1)$, and $(a', b') \leq_i (a, b)$, where $i = 1, 2$.

Based on this axiomatization of two-person games, we wrote a computer program (Tang and Lin 2011a) that generates many interesting theorems. It rediscovers Kats and Thisse's class of weakly unilaterally competitive games that correspond to the following condition:

$$(x_1, y) \leq_1 (x_2, y) \supset (x_2, y) \leq_2 (x_1, y) \wedge (x, y_1) \leq_2 (x, y_2) \supset (x, y_2) \leq 1(x, y_1).$$

It also returns some very strong conditions for strict two-person games to have unique Nash equilibrium, and led to some new theorems (Tang and Lin 2011b).

Again, this work can be easily cast into the general algorithm given earlier. The inputs can be described as follows:

Sorts: α and β.

Predicates: $\leq_1$ and $\leq_2$, both of type $(\alpha, \beta) \times (\alpha, \beta)$.

Collection of domains: just two elements for each sort, so for both sorts, their domains are $D = \{1, 2\}$.

Confirmation relation: a model is an interpretation in $D \times D$ that satisfies Σ, and a model confirms

a hypothesis φ if either sentence φ is not true in the model or the game corresponding to the model

has a unique Nash equilibria payoff. In other words, $M \models \varphi \supset (8)$.

Theorem prover: there is no need for a separate theorem prover for the hypotheses defined next. In this case, checking all models up to a certain size is sufficient.

Hypotheses: conjunctions of binary clauses.

Computer Program Verification

Computer program verification is a perfect application domain for machine theorem discovery. It is a very difficult but important problem given how crucial it is to have reliable software these days. It is also closely related to logic and theorem proving. There has been much work about it in software engineering and formal methods. An example system is Facebook infer1, a formal program analyzer initially developed by Monoidics in London and acquired by Facebook in 2013. It's said that all Facebook software has to go through infer before being released. Infer is based on separation logic (Reynolds 2002), an extension of Hoare's logic for reasoning about mutable data structures like pointers.

My colleagues and I are currently working on program verification based on a translation from programs to first-order logic that I proposed recently (Lin 2016). We have implemented a prototype system for programs with integer operations (Rajkhowa and Lin 2017). It makes use of off-the-shelf tools such as algebraic simplification system SymPy[2] and the SMT solver Z3 (de Moura and Bjorner 2012), and can already verify many nontrivial programs without user-provided loop invariants. It can also be extended to handle more complex data structures. In fact, it can easily be extended to handle arrays, and a version of it finished second in the array subcategory of the 2018 SV-COMP competition.[3]

However, to be able to handle more difficult programs, we need to have more effective proof strategies and this is where machine theorem discovery comes in. Consider the verification problem in a recent SV-COMP competition,[4] displayed in figure 2.

Effectively, our system (Rajkhowa and Lin 2017) will translate it into the problem of proving

$$a7(N_1) + b7(N_1) = 3 \times N$$

under axioms:

```
int i=0, a=0, b=0, n;
__VERIFIER_assume(n >= 0 && n <= 1000000);
while (i < n) {
if (__VERIFIER_nondet_int()) {
a = a + 1;
b = b + 2;
} else {
a = a + 2;
b = b + 1;
}
i = i + 1; }
__VERIFIER_assert(a + b == 3*n) ;
```

Figure 2. Verification Problem.

$$0 \le N \le 1000000,$$
$$\forall n.a7(n+1) = \text{ite}(f(n) > 0, a7(n)+1, a7(n)+2),$$
$$\forall n.b7((n+1)) = ite(f(n) > 0,\ b7(n) + 2,\ b7(n) + 1),$$
$$a7(0) = 0,$$
$$b7(0) = 0,$$
$$N1 \ge N,$$
$$\forall n.n < N1 \supset n < N,$$

where $N1$ is a natural number constant denoting the number of times the loop is executed before exiting; N the initial value of program variable ;, $f(n)$ an integer value function representing the nondeterministic function __VERIFIER_nondet_int() in the program; ite(c, e_1, e_2) the conditional expression "if c then e_1 else e_2"; and $\forall n$ ranges over natural numbers.

SMT solvers like Z3 (de Moura and Bjorner 2012) cannot prove this directly, but can be used to prove it from the following more general theorem:

$$\forall n.a7(n) + b7(n) = 3 \times n,$$

which can be proved by simple induction. The challenge is, of course, to formulate a space of hypotheses so lemmas like this can be discovered. I hope to have results to report on this in the near future.

Concluding Remarks

Theorem proving and theorem discovery are traditionally the job of mathematicians and theoreticians. Engineers just make use of discovered theories and theorems. However, in computer science at least, many "engineering" problems such as finding a plan to achieve a goal or writing an error-free program can be formulated as theorem proving, and one message of this article is that once a problem is formulated this way, machine theorem discovery becomes a part of the problem.

One may argue that machine theorem discovery is a type of machine learning. This depends on how broadly one defines machine learning. What should be clear is that machine theorem discovery is conceptually different from current machine learning problems. However, this does not mean that current machine learning techniques cannot be used to discover theorems. For example, to discover a closed-form formula $W(n)$ to make

$$\sum_{0 \le i \le n} i^2 = W(n)$$

true for all n, one can generate as many data as one needs — $W(0) = 0$, $W(1) = 1$, $W(2) = 5$, and so on — and then use a supervised learning algorithm to "learn" $W(n)$. However, this is not likely to work unless the target function $W(n)$ is very simple. It is clear that machine theorem discovery requires new tools and techniques. This is a rich area with many potential applications. I hope to have more results to report soon.

Acknowledgments

Foremost, I want to thank Yin Chen with whom I have collaborated on theorem discovery in logic programming, and Pingzhong Tang with whom I've collaborated on game theory. In particular, the work that I described above on two-person games was done jointly with Pingzhong. I also thank my other past and current collaborators on this project: Ning Ding, Jianmin Ji, Pritom Rajkhowa, and Haodi Zhang. I have also benefited from fruitful discussions on topics related to this work with Mordecai Golin, Jérôme Lang, Hector Levesque, Yidong Shen, Yisong Wang, Mingsheng Ying, Jiahua You, Mingyi Zhang, Yan Zhang, and Yi Zhou, among others.

Notes

1. fbinfer.com.
2. www.sympy.org/en/index.html.
3. sv-comp.sosy-lab.org/2018/.
4. v-comp.sosy-lab.org/2018/.

References

Brandl, F.; Brandt, F.; and Geist, C. 2016. Proving the Incompatibility of Efficiency and Strategyproofness via SMT Solving. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, 116–122. Palo Alto, CA: AAAI Press.

Brandl, F.; Brandt, F.; Geist, C.; and Hofbauer, J. 2015. Strategic Abstention Based on Preference Extensions: Positive

Results and Computer-Generated Impossibilities. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence*, 18–24. Palo Alto, CA: AAAI Press.

Brandt, F., and Geist, C. 2016. Finding Strategyproof Social Choice Functions via SAT Solving. *Journal of Artificial Intelligence Research* 55: 565–602.

de Moura, L., and Bjorner, N. 2012. The Z3 SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 337–340. Lecture Notes in Computer Science 4963. Berlin: Springer.

Fajtlowicz, S. 1988. On Conjectures of Graffiti. *Discrete Mathematics* 72(1–3): 113–118. doi.org/10.1016/0012-365X(88)90199-9

Friedman, J. 1983. On Characterizing Equilibrium Points in Two-Person Strictly Competitive Games. *International Journal of Game Theory* 12(4): 245–247. doi.org/10.1007/BF01769094

Geist, C., and Endriss, U. 2011. Automated Search for Impossibility Theorems in Social Choice Theory: Ranking Sets of Objects. *Journal of Artificial Intelligence Research* 40: 143–174.

Gerevini, A., and Schubert, L. 1998. Inferring State Constraints for Domain-Independent Planning. In *Proceedings of the 15th National Conference on Artificial Intelligence*, 905–912. Menlo Park, CA: AAAI Press.

Hoare, C. 1969. An Axiomatic Basis for Computer Programming. *Communications of the ACM* 12(10): 576–580.

Huang, Y.-C.; Selman, B.; and Kautz, H. A. 2000. Learning Declarative Control Rules for Constraint-Based Planning. In *Proceedings of the 17th International Conference on Machine Learning*, 415–422. San Francisco: Morgan Kaufmann.

Kats, A., and Thisse, J. 1992. Unilaterally Competitive Games. *International Journal of Game Theory* 21(3): 291–299. doi.org/10.1007/BF01258280

Larson, C. E. 2005. A Survey of Research in Automated Mathematical Conjecture-Making. In *Graphs and Discovery*, 297–318. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 69. Providence, RI: American Mathematical Society.

Larson, C. E., and Cleemput, N. V. 2016. Automated Conjecturing I: Fajtlowicz's Dalmatian Heuristic Revisited. *Artificial Intelligence* 231: 17–38. doi.org/10.1016/j.artint.2015.10.002

Larson, C. E., and Cleemput, N. V. 2017. Automated Conjecturing III: Property-Relations Conjectures. *Annals of Mathematics and Artificial Intelligence* 81(3-4): 315–327. doi.org/10.1007/s10472-017-9559-5

Lenat, D. B. 1979. On Automated Scientific Theory Formation: A Case Study Using the AM Program. In *Machine Intelligence 9,* edited by J. Hayes, D. Michie, and L. I. Mikulich, 251–283. Chichester: Ellis Horwood

Lin, F. 2004. Discovering State Invariants. In *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning*, 536–544. Palo Alto, CA: AAAI Press.

Lin, F. 2007. Finitely-Verifiable Classes of Sentences. In *Logical Formalizations of Commonsense Reasoning: Papers from the 2007 AAAI Spring Symposium*. Technical Report SS-07-05. Palo Alto, CA: AAAI Press.

Lin, F. 2016. A Formalization of Programs in First-Order Logic with a Discrete Linear Order. *Artificial Intelligence* 235: 1–25. doi.org/10.1016/j.artint.2016.01.014

Lin, F., and Chen, Y. 2007. Discovering Classes of Strongly Equivalent Logic Programs. *Journal of Artificial Intelligence Research* 28: 431–451.

Monderer, D., and Shapley, L. S. 1996. Potential Games. *Games and Economic Behavior* 14(1): 124–143. doi.org/10.1006/game.1996.0044

Moulin, H. 1976. Cooperation in Mixed Equilibrium. *Mathematics of Operations Research* 1(3): 273–286. doi.org/10.1287/moor.1.3.273

Rajkhowa, P., and Lin, F. 2017. VIAP: Automated System for Verifying Integer Assignment Programs with Loops. In *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Reynolds, J. C. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science: Proceedings of 17th Annual IEEE Symposium,* 55–74. Piscataway, NJ: IEEE. doi.org/10.1109/LICS.2002.1029817

Tang, P., and Lin, F. 2009. Computer-Aided Proofs of Arrow's and Other Impossibility Theorems. *Artificial Intelligence* 173(11): 1041–1053. doi.org/10.1016/j.artint.2009.02.005

Tang, P., and Lin, F. 2011a. Discovering Theorems in Game Theory: Two-Person Games with Unique Pure Nash Equilibrium Payoffs. *Artificial Intelligence* 175(14–15): 2010–2020. doi.org/10.1016/j.artint.2011.07.001

Tang, P., and Lin, F. 2011b. Two Equivalence Results for Two-Person Strict Games. *Games and Economic Behavior* 71(2):479–486. doi.org/10.1016/j.geb.2010.04.007

Topkis, D. 1998. *Supermodularity and Complementarity*. Princeton, NJ: Princeton University Press.

Wang, H. 1960. Toward Mechanical Mathematics. *IBM Journal of Research and Development* 4(1): 22–22. doi.org/10.1147/rd.41.0002

Fangzhen Lin (PhD, Stanford University) is a professor of computer science at the Hong Kong University of Science and Technology. He is an AAAI fellow.