Designing Games of Theorems

Who am I?  What do I like?  What did I develop?

Yutaka Ng  yutakang
Block or report user

automating proof search
in expressive logic (HOL)
using heuristics / ML

PSL/PaMpeR for Isabelle/HOL
Proof Strategy Language (PSL) for Isabelle/HOL

- meta-tool approach
- tactics
- quickcheck
- sledgehammer
- runtime tactic generation
- efficient proof generation
- native Isabelle proof script
- easy installation

- no code clutter!!

- extensive proof search
- parallel search
- low memory usage
- extensible (Eisbach)

- programming language

- PSL

Yutaka Ng
yutakang
CVUT, CTU, CIIRC
try_hard: the default strategy

strategy Basic =
Ors [
    Auto_Solve,
    Blast_Solve,
    FF_Solve,
Thens [IntroClasses, Auto_Solve],
Thens [Transfer, Auto_Solve],
Thens [Normalization, IsSolved],
Thens [DInduct, Auto_Solve],
Thens [Hammer, IsSolved],
Thens [DCases, Auto_Solve],
Thens [DCoinduction, Auto_Solve],
Thens [Auto, RepeatN(Hammer), IsSolved],
Thens [DAuto, IsSolved]]

strategy Try_Hard =
Ors [Thens [Subgoal, Basic],
    Thens [DInductTac, Auto_Solve],
    Thens [DCaseTac, Auto_Solve],
    Thens [Subgoal, Advanced],
    Thens [DCaseTac, Solve.Many],
    Thens [DInductTac, Solve.Many]]
try\_hard vs sledgehammer

The percentage of automatically proved obligations out of 1526 proof obligations
(timeout = 300s)

- try\_hard: 73%
- sledgehammer: 57%
try_hard: the default strategy

strategy Basic =
  Ors [
    Auto_Solve,
    Blast_Solve,
    FF_Solve,
    Thens [IntroClasses, Auto_Solve],
    Thens [Transfer, Auto_Solve],
    Thens [Normalization, IsSolved],
    Thens [DInduct, Auto_Solve],
    Thens [Hammer, IsSolved],
    Thens [DCases, Auto_Solve],
    Thens [DCoinduction, Auto_Solve],
    Thens [Auto, RepeatN(Hammer), IsSolved],
    Thens [DAuto, IsSolved]]

strategy Try_Hard =
  Ors [Thens [Subgoal, Basic],
        Thens [DInductTac, Auto_Solve],
        Thens [DCaseTac, Auto_Solve],
        Thens [Subgoal, Advanced],
        Thens [DCaseTac, Solve_Many],
        Thens [DInductTac, Solve_Many]]
preprocess

decision tree construction

fast feature extractor

large proof corpora

feature vector

database

:: ( tactic_name, [ bool ] )

lookup

proof method

recommendation

PaMpeR

proof

state

proof engineer

preparation phase

recommendation phase

full feature extractor

feature vector

proof meta recommendation

DEMO!
from try_hard to try_smart

PSL & try_hard: more computation

PaMpeR: get smart using heuristics
Higher-Order functions, polymorphism, universal quantifier, lambda abstraction, type class, dependent types, concise formula that can cover lots of concrete cases, small data set for each problem, different proof for general case.
Large Proof Corpora?

The Kepler “conjecture” in HOL Light

Four color theorem in Coq

The seL4 proofs in Isabelle

OpenTheory Project

Goldbach’s conjecture

“Every even integer greater than 2 can be expressed as the sum of two primes.”

different logics in different provers
different proof corpora about different problems

Transfer learning? ✗ (?)
poor proof automation for expressive logics

only small dataset available because of expressiveness

artificial intelligence for theorem proving!

We need big data!

really?
I want to train my prover using self-play so that it can prove Goldbach’s conjecture. But how? Proof search is not a 2-player game.

The one that finds a proof of Goldbach’s conjecture first is the winner. If one prover finds a proof, that’s it. It is only 1 iteration.

But how do you train provers, so that one prover can eventually find a proof.

For each iteration, I create a set of not-so-difficult conjectures. The one that proves more conjectures is the winner.

But how do you create not-so-difficult conjectures?

random?

But randomly created conjectures are not always good training data.

Conjectures with difficult proofs are important ones. Not really. You need a mechanism to create many conjectures that are relevant to Goldbach’s conjecture.

I can produce conjectures by mutating Goldbach’s conjecture. That might work for a small number of conjectures. Not for many conjectures.

How?

The more conjectures you create, the more valuable they should be.
I want to train my prover using self-play so that it can prove Goldbach's conjecture. But how? Proof search is not a 2-player game. The one that finds a proof of Goldbach's conjecture first is the winner. If one prover finds a proof, that's it. It is only 1 iteration. But how do you train provers, so that one prover can eventually find a proof.

For each iteration, I create a set of not-so-difficult conjectures. The one that proves more conjectures is the winner. But how do you create not-so-difficult conjectures? random? But randomly created conjectures are not always good training data. Conjectures with difficult proofs are important ones. Not really. You need a mechanism to create many conjectures that are relevant to Goldbach's conjecture.

http://cl-informatik.uibk.ac.at/teaching/ss18/mltp/02.pdf

Research hypothesis: subgoals proved during heuristic (incomplete) proof search are useful to train provers.

Game of Theorems!

The more iterations it goes through, the higher the quality of problems should be!

...the more conjectures you create, the more valuable they should be.
Game of Theorems 1

Coq vs Parrot vs Crow

proved subgoals  search tree  big conjecture

Coq

Parrot

Crow

70% 80% 60%

different problems for different prover?
Game of Theorems 2

proved subgoals  search tree  big conjecture

Owl vs Crow

only one prover can survive?

70%  80%
Game of Theorems 3

70% -> Coq vs Parrot vs Crow

60% -> What if

80% -> ?
Game of Theorems 4

No ordering, no casualty.
Research hypothesis:
subgoals proved during heuristic (incomplete) proof search are useful to train provers.

No ordering, no casualty.
Thanks,