

600.425 — Declarative Methods (graduate version)

Term Projects

Spring 2009
Prof. J. Eisner

Proposal due: Friday, Apr 3 (≤ 1 page, not graded)
Project due: Sunday, May 10 (3 days before our exam)

The graduate version of Declarative Methods, 600.425, includes a term project (25% of your grade). You may work in pairs, although a 2-person project should be somewhat more ambitious than a 1-person project.

The main thing to hand in is a paper (perhaps 8–10 pages) that describes (1) the problem you decided to solve, (2) how you went about solving it, and (3) the results. Generally the results would include an experimental comparison or a theorem. You only have a month or so, but you may want to use this project as a starting point for publishable work and/or a degree project.

Please email a project proposal (≤ 1 page, preferably in PDF) to cs325-staff@cs.jhu.edu by next week. Anything that is related to the course is within bounds. But you should probably discuss the topic and scope of your project with me before writing the proposal, to make sure that it is both interesting and feasible. Feel free to email your thoughts or set up a meeting.

Since your project shouldn't just duplicate past work, you ought to check the web carefully to see if anything related has been done. Describe in your proposal what you found (a bibliography would be appropriate) and what your new contribution will be. If there's previous work, that's good because you can build on other people's existing software or experimental comparison.

Some general advice on finding research problems is at <http://www.cs.jhu.edu/~jason/advice/how-to-find-research-problems.html>. For this class, there are at least four or five obvious kinds of topics:

1. You could demonstrate that existing declarative methods provide an effective solution to some real-world problem that is of interest to you.
2. You could devise (and try out) a new technique for making an existing solver more efficient or accurate.

3. You could extend an existing declarative language to handle a wider class of problems (or handle the same class more elegantly), and figure out how to improve the solver to handle this extended language.
4. For a given type of problem that is of interest to you (not necessarily NP-hard), you could theoretically or experimentally compare the performance of different approaches.
5. You could create your own “little language” for some problem of interest to you.

Here are a few thoughts that come to mind, but please don't feel limited by these:

1. Solve a real-world problem
 - Anything in your research specialty
 - Intelligent playing of a board game¹
 - Register allocation, instruction scheduling, etc.
 - Large-scale crossword construction (as in HW1); or solving crossword puzzles that have clues
 - Task scheduling (e.g., the RCPS benchmarks from HW2)
 - Handwriting recognition (e.g., as in HW3)
 - Other industrial tasks, e.g., airline route planning
 - Scheduling sports matches (MLB, NFL) constrained by league rules, TV demands, audience size estimates, travel cost, etc.
 - Graph layout (the 2005 midterm had a problem about this)
 - Bioinformatics
 - Grammar learning
2. Make an existing solver more efficient or accurate
 - Show how to train neural nets, or learn finite-state grammars, using stochastic local search
 - Implement a better stochastic SAT solver (UBCSAT makes this “easy”), perhaps using statistical or other techniques (e.g., John Blatz suggested incorporating EM)

¹*Warning:* Game playing is *not* a good idea for SAT/constraint solvers, since it's basically QSAT, not SAT. But machine learning of some kind might help pick a good single move. Perhaps even finding a legal move could be challenging in some settings (Scrabble?).

- Implement a better systematic SAT solver (e.g., Mark Miller had a suggestion about improving zChaff)
 - Design a better variable or value ordering strategy, perhaps with more learning
 - Combine dynamic programming with constraint programming or stochastic search (e.g., various NLP problems; graph layout)
 - Better portfolio methods (e.g., use machine learning to pick which solver and parameters to use for a given problem)
3. Extend an existing declarative language or its solver
- Improve Dyna in any useful way
 - Add finite-state constraints to ECLiPSe
 - Design new constraint propagators (for a standard constraint or a new constraint you propose) and show that they lead to faster solutions
 - Define a new SVM kernel and show that it beats the existing ones
 - Linear functions for choosing best or near-best element from a set (rather than classifying good vs. bad; ask me about this)
 - Design a non-CNF-SAT solver that does something smarter than just convert to CNF first (this has been done for systematic but not stochastic solvers, as far as I know)
 - QSAT (SAT of quantified boolean formulas)
4. Comparisons on a problem of interest to you
- Is one solver likely to work better than another? Does it?
 - Is one encoding likely to work better than another? Does it?
 - Are declarative approaches competitive with standard, specialized algorithms for this problem? If so, can you prove it? If not, how could the declarative approaches be improved to handle this class of problems more efficiently?
5. Design your own “little language” and solver. This could be anything from your research or personal interests. However, for it to be an appropriate project, either the language design or the solver has to raise interesting issues.

I look forward to hearing about your ideas!!

In case you're interested, here were the project topics from 2008:

- generating Sudoku puzzles of a given level of difficulty (i.e., puzzles that will force a given solver to backtrack)
- a little language for describing card games and strategies for playing them, with a “solver” that plays those games against a human
- a machine learning classifier that uses a decision tree to partition the input space and an SVM to classify within each partition
- an enhanced type system for the Dyna programming language

And the project topics from 2007:

- adding finite-state constraints to ECLiPSe
- contrapuntal harmonization and fugue exposition composition using constraint logic programming
- efficient crossword solver (written directly in C++, using various constraint programming techniques and implementation tricks)
- protein folding
- solving Boggle using ECLiPSe

And the project topics from 2006:

- improve handwritten digit recognition (as in HW3) by distorting the training examples to produce new training examples
- instruction scheduling for a pipelined processor (tried both ECLiPSe and Integer Linear Programming)
- improvise blues melodies through machine learning
- find a regular expression that accounts for a training set of example strings (used a genetic algorithm)
- find a good fingering (i.e., easy to play) for a guitar melody
- given a class of SAT problems, learn a variable ordering strategy (used a genetic algorithm to learn a weight vector)
- schedule a season’s worth of National Football League games (a constraint programming problem, but it’s hard to achieve a solution in reasonable time)
- same thing, but for Major League Baseball games
- play Othello (Reversi) against a human
- play poker against a human
- help a human play Civilization IV, using a little language and a solver written in Dyna
- discover variant spellings of Arabic words
- modify the WalkSAT algorithm to work better on DIMACS challenge problems