CSci 4511W Final Project "Effort Paper"

Sean Bowman

May 4, 2011

As I am the sole member of this project group, I of course did all work presented in the real paper. I implemented everything described from scratch in Python, using no third party libraries or other peoples' code. A description of each file in the system follows:

- LogicTokenizer.py
  - Tokenizes an input string, converting a set of characters to a more computer friendly form. Example: "a => b" becomes [Token('var', 'a'), Token('implies'), Token('var', 'b')]
- LogicParser.py
  - Parses an input string with the help of LogicTokenizer, converting any input expression into an Expression class representing its meaning.
- LogicSimplifier.py
  - Several methods to simplify a logical statement, culminating in to_cnf() which converts any input Expression into conjunctive normal form.
- KnowledgeBase.py
  - The actual resolution system, taking input from users and, through the help of LogicSimplifier and LogicParser, acts as a functioning knowledge based as described in the main paper.
- testlogic.py
  - My little test bed for the knowledge base. Reads several test cases from "tests.txt" (the syntax of that text file is pretty intuitive), timing how long it takes to prove the desired result and ensuring the correctness of that result.

I believe I presented the work done clearly in the real paper, however I will go over several bullet points here:

- Implementing the parsing system

  - I initially tried to implement unnecessarily complex parsing algorithms, i.e. a recursive descent parser, the CYK algorithm, etc. I eventually realized that I can make a simple infix parser work with a few modifications.
  - To do this I had to learn about regular expressions, parsing of languages, etc. Took me a while to do actually.

- Logic simplification algorithms
  - This was the brunt of my work, I believe. Even once I figured out how to conveniently express logical sentences in Python (my final solution was a single "Expression" class, whose actual meaning varies based on its contents), it was quite a bit of work translating logical entailment theorems into code.
  - What complicated things is that everything has to be done recursively: e.g. once implications are eliminated at one level, you have to recursively call the function on each of the sub-expressions as well.
  - The most difficult rule to implement was the distribute_or_over_and function. Handling a case such as $a \vee (b \wedge c)$ was very simple, however the complexity needed to increase by quite a bit to handle cases like $(a \wedge b) \vee (c \wedge d)$ appropriately. The final result handles all cases well, no matter how large. $(a \wedge b \wedge c \wedge d) \vee (e \wedge f \wedge g \wedge h) \vee (i \wedge j \wedge k \wedge l)$ is no match for my algorithm (the result is a beast containing 64 clauses).

- Resolution System
  - This was a bit tricky to get to work correctly, though once everything was in CNF form from the logic simplification functions this wasn't terrible.
  - Implementing the resolution algorithms did cause me to shift between representations before I settled on what was most convenient, though. Initially I tried representing clauses as their actual logical expressions, e.g. (a or b or c). After trying to work with that for a bit, I realized it was needless and cumbersome, and decided to convert them to sets before resolving, e.g. $\{a, b, c\}$.

Nearly all of this project was completely new to me (parsers, regular expressions, logic programming of any sort, etc.), and I learned a lot in the process of creating this system.