Automated Reasoning

By Alec Rajeev

CSC 242

Ferguson

I. Introduction

The purpose of this project was to implement two techniques that utilize Propositional Inference. The first technique used is basic model model checking using Truth Table Enumeration. The second second technique used was the WalkSAT algorithm. These techniques were used to for four main problems: Modus Ponens, Simple Wumpus World, Horn Clauses, and Liars and Truth Tellers. The programming language used was Python 2.7, and the numpy library was used as with it.

II. Propositional Logic Structure

The overall structure of my program was influenced from the Java code that Prof. Ferguson gave to the class. The first main class was Knowledge Base class. This had three major components: a python list of sentences, a Hash Table of symbols, and a python list of models. The sentences were their own class and were separated between Complex Sentences and just Sentences. Only Complex Sentences were added to the list in the Knowledge Base. A Complex Sentence would have a left hand side and right hand side if it was a more complicated clause. If it was a Complex Sentence was just a single clause, then it would include just a single Sentence object. An integer attribute grammar_type of Complex Sentence

was used to determine what type it was. A simple positive clause would be type 0. A simple negation clause would be type 1. Then it moves onto the aforementioned binary sentences such as conjunction, disjunction, implies, and biconditional. The type of these increases in such a way to satisfy the order of operations. This method of using a Complex Sentence object to split apart larger sentences is a way of describing structures without inheritance. I found this to be the most simple way to describe things. For example, a Complex Sentence that is a biconditional can have the right hand side sentence be another Complex Sentence that is a conjunction. This made it possible to write any propositional logic statement in my program. A component of a simple Sentence object is an atom that gives the name of the symbol that it represents such as “p”. Every Complex Sentence is made up of one or more of these eventually.

The representation of a model will be described in more depth below, but it is essentially a set of possible truth values for the symbols. The Sentence object has a method called isSatisfiedBy that takes a model object, and returns whether that sentence is satisfied by the model. This is also influenced from Prof. Ferguson’s design. A Complex Sentence used a slightly longer version of this that recursively calls the isSatisfiedBy method on its left hand side and right hand side parts. It also combines the parts depending on what the grammar_type is (conjunction, disjunction, etc.). Eventually a simple sentences isSatisfiedBy method is called.

Similar to Ferguson’s structure, all of the symbol’s are put into a Hash Table. The key of the symbol is it’s name like “P(1,2)”. The value that comes out of the Hash Table is a randomly chosen integer that represents the Symbol’s id. Thus each symbol gets its own integer id. The self designed hash function is simple and

involves converting the characters of the symbol to their ascii number and multiplying them. Separate chaining is used to handle collisions; although, there are generally so few characters that it is not really necessary. The Node object is used to store a piece of information in the Hash Table object. This Hash Table is an attribute of the Knowledge Base.

A Model Table object is built from symbol list. Suppose there are n number of symbols. Then there must be 2^n number of possible models assuming that each symbol can be only true or false. The model table has for the first n columns be the symbol's id number. The second n columns are all the possible values for the symbols to give all the models. There are 2^n rows where each row is its own model. This is a 2-d numpy array of ints. Then once the Model Table object is built, it is converted to a python list of Model object. Each model object represents a single model and includes the symbol's true or false values and the symbol's integer id. Splitting it up this way made it easier to pass through the model object to various methods. These list of models is an attribute of the Knowledge Base.

II. Basic Model Checking

Now the basic model checking using truth table enumeration will be used. After the symbols are entered into the Knowledge Base (uses the method intern like Prof. Ferguson), and the sentences are also added to the Knowledge Base, the models are built. Using a combination of modulo 2 and powers, all possible models are built and stored in a python list called model_list. Then the Knowledge Base object for each model, loops through every sentence. Using the isSatisfiedBy method, it checks if a particular model satisfies that sentence. The sentence stores a symbol like "P(1,2)", so the Hash Table is used to convert this symbol to an

integer id. This connects it to a particular model. This was done in case larger data sets were implemented in the future to improve the speed. If every sentence satisfies a particular model, then the index of that model in the list is stored. A different list keeps track of the index of every model that satisfies all of the sentences. Then the knowledge base finds all of the models that satisfy a particular alpha. The alpha is a sentence. The Knowledge Base object builds a list of indexes of every model that satisfies the alpha. Then it compares the two lists of indexes. If the list of indexes of models that satisfies the knowledge base is a subset of the list of indexes that satisfy the alpha, then the knowledge base entails alpha. However, if there is a model that satisfies the knowledge base, but does not satisfy the alpha, then it is inconclusive. This describes how the Truth Table enumeration method is used to check for entailment.

III. WalkSAT

Now the WalkSAT algorithm will be used to check for entailment. The first thing that is done is the algorithm randomly chooses a model. Then it checks if the model satisfies all of the sentences in the knowledge base. If the model satisfies everything then it returns true, and states that at least one model satisfies all of the sentences. As it is looping through the sentences it stores in a list the index of sentences that are not satisfied by the model. If a model is unsatisfactory, it will randomly choose a sentence that is not satisfied from the stored list. Then it will do a different thing depending on the random float is less than p (usually .5). If it is less than .5, then a randomly selected symbol in the clause that was chosen will be flipped. Then it loops through everything again. If the random float is not less than .5, then it will flip whichever symbol in the previously selected clause that will overall

maximize the number of sentences/clauses. Then it will loop through everything again using the model with the flipped symbol.

When implementing the WalkSAT algorithm for entailment, one will add the input sentence that is the negation of what you are trying to show is entailed from the knowledge base. The WalkSAT algorithm will keep searching for a model that satisfies all of the sentences. If a model is found, then it is inconclusive and you can not say anything about the entailment. However if after a sufficiently large number of trials (I chose 10,000), no model is found then one can say with a reasonably good probability that the knowledge base entails alpha. However, this is not a formal proof because the WalkSAT would have to run an infinite number of times for it to work formally. This is impossible, but one can conclude with likely probability of entailment.

IV. Conclusion

These two techniques were used to show entailment for 5 separate problems. Although the problems were small, they solved everything in a reasonable amount of time, often under thirty seconds if not quicker. While the WalkSAT method is not a formal proof, for large problems it is often a quick way of finding a model that satisfies the solution. Finding a satisfied model is inconclusive on entailment however. The truth table enumeration was a simple way to check one's work. It is very intuitive and I decided not to use the lisp-like algorithm given in the book.