Project report on

Belief Revision

for 02180 Introduction to Artificial Intelligence

Contributors:	Jannis Haberhausen	(s186398)
	Jack Reinhardt	(s186182)
	Kilian Speiser	(s181993)
	Jacob Miller	(s186093)

Contents

1 Introduction

2 Design and Implementation

The design of the belief base takes advantage of the object-oriented nature of the Python programming language. At the highest level, the contents of the belief base are stored in an instance of a class called BeliefBase. The BeliefBase class simply contains a list of beliefs along with methods for adding or removing beliefs, checking for logical entailment, performing partial meet contraction, and showing the current belief base. One level lower, the beliefs are represented as an instance of the class Belief. Each belief contains a list of clauses, which is initialized when the object is created, taking an input string in CNF with an option for negating the entire belief. The Belief class also contains methods to print the belief or to convert the belief to a string. At the lowest level is the Clause class, which stores a list of the positive propositional symbols and a list of the negative propositional symbols. These two lists are initialized when each Clause is created by parsing the string that is input. The methods in Clause include deleting symbols, checking if the clause is empty and printing or converting the clause to a string. The Clause class also implements static methods to combine clauses, create a copy of a clause, check if two clauses are equal, negate a clause, and resolve two clauses.

2.1 Implementation - Converting to CNF

As a pre step to later actions, like belief revision and partial meet contraction, we implemented a function that could turn propositional logic into its coresponding conjunctive normal form (CNF). A sentence in CNF consists of multiple clauses connected with AND ('∧') symbols. The order in which we turn propositional logic into CNF is as follows

$$\text{Eliminate BICONDITIONALS } ('\leftrightarrow') : \qquad p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$$
$$\text{Eliminate IMPLICATIONS } ('\rightarrow') : \qquad p \rightarrow q \equiv \sim p \vee q$$
$$\text{Perform DE MORGAN} : \qquad \sim (p \vee q) \equiv (\sim p \wedge \sim q)$$
$$\sim (p \wedge q) \equiv (\sim p \vee \sim q)$$

Figure 1: Steps to convert propositional logic into CNF

Finally, the OR ('∨') symbols are distributed over the AND ('∧') symbols in order to derive the CNF. Each of the above steps are performed as many times, until there are no more instances of the decribed operators, or until a sentence in CNF is derived. This iterative approach ensures that the sentence will be transformed into CNF, independend of the number of propositional operators, or if it is in disjunctive normal form (DNF), or any other logical form. At this point, it is mentioned, that the use of parenthesis is important for the transforming algorithm to work properly, e.g $p \rightarrow q \rightarrow s$ might cause problems, while $(p \rightarrow q) \rightarrow s$ will result in the correct CNF. In other word, it needs to be ensured that each binary operator together with its two arguments is enclosed by parenthesis. By default, we insert parenthesis around the whole input in order to make the writing more convenient for the user, e.g $(p \rightarrow q) \leftrightarrow (s \wedge t)$ is a valid input.

The input sentence in propositional logic is stored as a string. In the next steps the string is checked if it contains any BICONDITIONAL ('↔') or IMPLICATION ('→') symbols. If it does the algorithm starts to eliminate them. It does that by dividing the sentence into three different parts, all stored in separate strings. The first part contains the binary operator together with its two arguments. The other two parts contain the rest of the original sentence that is not affected by the operator, to the left and and to the right, respectively. This is especially convenient, because the same function can be used for solving BICONDITIONALS and IMPLICATIONS. Finally the algorithm sets the three strings back together to one string, after eliminating the original operator and replacing it by its correct simplification shown in figure 1. The function that simplifies sentences using the De Morgan rules, works differently, since there is no operator that divides the two arguments that are affected by this operator. Also notice that

De Morgan can be applied to multiple literals at the same time, eg. $\sim (p \land q \land t) \equiv (\sim p \lor \sim q \lor \sim t)$. First a given sentenced is checked if it contains any instances where a negation ('$\sim$') is directly followed by a parenthesis ('('). If this is the case, the de Morgan algorithm is applied to the sentence (notice that it is invalid to write single literals in parenthesis). The algorithm keeps track of the parenthesis that are used within the part that de Morgan is applied to, eg. $\sim ((p \lor q) \land t) \equiv (\sim (p \lor q) \lor \sim t)$. Notice that in this case the de Morgan algorithm will be applied to the resulting sentence again. After all BICONDITIONALS and IMPLICATIONS, and instances where the de Morgan rule can be applied, are eliminated, it is checked if the sentence contains any double negations ('$\sim\sim$') that can be erased. Finally the OR symbols are distributed over the AND symboly to derive the final CNF.

2.2 Implementation - Converting CNF into Beliefs and Clauses

Once the input string is converted into CNF, the string is converted into a list of clauses, which is stored in a belief. Each AND ('$\land$') symbol marks the end of a clause when in CNF; any spaces, double negations, and parenthesis are removed from the string before it is used to initialize a new clause object. The initialization of each clause then removes all OR ('$\lor$') symbols and adds each propositional symbol to either the positive symbol list or the negative symbol list (symbols that are negated).

2.3 Assumptions

The design and implementation of the belief base makes one important assumption: that each propositional logic symbol is a single character. This was a design choice for robustly implementing the parsing of string and converting the strings into beliefs and clauses.

3 Logical Entailment

A resolution-based approach was used to check for logical entailment in the belief base. Once a belief base has been initialized and beliefs have been added to it, logical entailment can be checked of an input belief in string form. First, the input string is converted into its negation as a belief object as described in section 3.2. Then, all of the clauses in the belief base and the clauses of the input belief are stored in a single list to iterate over. The resolution algorithm loops over pairs of clauses, resolves these clauses, and then checks if the resolvent is either empty or already in the list of clauses. The implementation of the resolution algorithms is further explained in section 3.1. If two clauses resolve to an empty clause, then the method returns true, indicating that the input belief is entailed by the belief base. Otherwise, if the list of clauses has been exhausted and no further clauses can be added through resolution, then the loop terminates and the method returns false, indicating that the input belief is not entailed by the belief base.

A list containing all pairs of clauses that have already been resolved was added to improve the efficiency of the resolution algorithm. Each time the algorithm iterates over a pair of clauses, it checks that the clauses are not equal (not the same clause) and that the pair of clauses is not in the list of previously resolved clauses.

3.1 Resolution

A static method was implemented to resolve two disjunctive clauses in the Clause class. The algorithm first searches for and removes any complementary literals in the two clauses. Next, it searches for and removes any redundant symbols (i.e. 'avbvb' would simplify to 'avb'). Finally, the two clauses are combined into a single disjunctive clause and returned as the resolvent.

3.2 Belief Negation

When checking whether or not a belief is entailed by the belief base, resolution essentially performs a proof by contradiction, which requires negating the input belief. To negate the belief, it is first broken up into individual clauses. Each clauses is then negated through the use of De Morgans Law. Distributing

$\vee$ over $\wedge$ then yields the negated belief in CNF, which can be used to check for logical entailment with resolution.

4 Revision

4.1 Partial Meet Contraction

5 What We've Learned

6 Conclusion and Future Work