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

Belief revision is a necessary component of any artificially intelligent system that plans to adapt its knowledge as new inputs come in. AGM belief revision theory consists of three methods of adapting a belief base: expansion (+), contraction ($\div$) and revision (*). Levi's identity shows how to build a general algorithm for revision using only expansion and contraction since $B * p = (B \div p) + p$. In this report, the method of partial meet contraction will be described in detail in section 4. The belief base will check for logical entailment of beliefs with a resolution-based method as described in section 3, which is utilized when performing belief contraction. Specific implementation details will be described throughout the report.

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 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 ('$\wedge$') symbols. The order in which we turn propositional logic into CNF is as follows

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

Figure 1: Steps to convert propositional logic into CNF

Finally, the OR ('$\vee$') symbols are distributed over the AND ('$\wedge$') 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 ('$\leftrightarrow$') or IMPLICATION ('$\rightarrow$') 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. $\sim (p \wedge q \wedge t) \equiv (\sim p \vee \sim q \vee \sim t)$. 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. $\sim ((p \vee q) \wedge t) \equiv (\sim (p \vee q) \vee \sim t)$. 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.

Disclaimer: We unfortunately noticed, that De Morgan is not resulting in the correct form, which means that also the CNF form is not correct for all inputs in propositional logic.

2.2 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 ('$\wedge$') 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 ('$\vee$') 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

Revising a belief base with φ involves contracting $\neg\varphi$ from the belief base and an expansion of φ into the belief base. This follows from Levi's Identity:

$$B * \varphi := (B \div \neg\varphi) + \varphi$$

This is precisely the method used in the implementation of revision. If contraction and expansion are implemented correctly, revision follows with little to no effort. Expansion simply involves adding a belief to the belief base without care for potential contradictions in the belief base. In other words, φ is added to belief base B giving a new belief base B'. Appending another belief to the internal belief list of the belief base solves this problem. Contraction is more involved and explained in the coming sections.

4.1 Contraction

Contraction is defined as removing φ from the belief base B giving a new belief base B'. After contraction, φ must not be entailed by the belief base. This is where remainder sets become useful. They are defined as the set of inclusion-maximal subsets of B that do not entail φ and are denoted $B\perp\varphi$. The contraction could be any one of these remainders, but there are a few different methods for choosing which remainder set to return. These methods are detailed after a discussion on how the belief revision engine calculates inclusion-maximal remainder sets.

4.1.1 Remainder Sets

The belief revision engine calculates remainder sets using a variation of Breadth First Search - Backwards Clause Selection. It is similar to Backwards Feature Selection in machine learning contexts. When calculating remainder sets for $B\perp\varphi$ the algorithm removes a belief from the B and checks if the resulting belief base entails φ. It checks all possible remainders of size $|B| - 1$, then all possible remainders of $|B| - 2$, down to the empty remainder set $\varnothing$. Formally, the search problem is defined as follows:

s_o	initial state
$ACTIONS(s)$	from a belief base s, removing each belief
$RESULTS(s, a)$	resulting belief base after removing a belief
$GOAL - TEST(s)$	test if s entails φ
$STEP - COST(s, a)$	step cost is 1

Instead of returning the first solution we find, we return all of the solutions of that length, so we have a list of remainders. Now that the remainder sets are calculated, the different methods for completing contraction can be explained. They include maxichoice contraction, full-meet contraction, and partial-meet contraction.

4.1.2 Maxichoice Contraction

The simplest method to choose remainders is using Maxichoice contraction. This method involves picking a random remainder and returning it. The belief revision engine always returns the first remainder calculated by the Backward Clause Selection algorithm. This method is not the best, as it does not take into account any other remainder.

4.1.3 Full-Meet Contraction

Another simple method to choose remainders is to take the intersection of all remainder sets. If R is the set of remainders, contraction returns $\bigcap_{i=1}^{|R|} R_i$. The belief revision engine does just this, and intersects all remainders calculated by the Backwards Clause Selection algorithm. This is also not an ideal solution, as the result of contraction is much too small to be useful.

4.1.4 Partial-Meet Contraction

A more complex method to choose remainders is to take the intersection of *some* of the remainder sets. The interesting aspect of partial-meet contraction is how the remainder sets are chosen. Each belief in the original belief base is assigned a entrenchment value. This value should reflect how deeply this belief is held in the belief base and satisfy the requirements for Epistemic Entrenchment. The difficult aspect of partial-meet contraction is assigning values to each belief so that we have a valid Epistemic Entrentchment scheme, denoted by $\leq$. Once these values are assigned though, the belief revision engine computes all possible remainder intersections. From these possible contraction results, the engine finds the intersection that has the maximum entrenchment sum of the beliefs. This implies that the result of contraction will have the highest possible total entrenchment value, and therefore favor the deepest held beliefs over weaker held beliefs.

The belief revision engine assigns values to beliefs based on if one belief is entailed by another and if a belief is contained in a conjunction of another belief. φ and ψ are beliefs. If $\varphi \models \psi$, then $\varphi \leq \psi$ and the entrenchment value of ψ increases by 1. If $\varphi \in B$ and $\varphi \wedge \psi \in B$, then $\varphi \leq \varphi \wedge \psi$ and the entrenchment value of $\varphi \wedge \psi$ increases by 1. After this algorithm is run on the uncontracted belief base, we have a valid ordering satisfying the 5 requirements from epistemic entrenchment. We then square the entrenchment values in order to assign more value to deeper held beliefs. This comes into effect when we calculate the maximum entrenchment sum of the intersected beliefs.

4.2 Revsion

Now that the belief revision engine has the capability to perform contraction of the three types listed in section 4.1, it is capable of performing revision. The belief revision engine takes a belief base, φ, and a contraction method. The engine then negates φ and performs contraction of the specified method of $\neg\varphi$ on B. After this has completed, B is simply expanded by φ.

5 What We've Learned

Throughout the project we learned more about belief revision and entailment. We learned about different ways to choose the remainder sets to intersect in partial-meet contraction - specifically epistemic entrenchment and its properties. There is a lot of freedom when choosing sets to intersect in partial-meet contraction, and there are likely many creative ways to yield great contraction solutions. Finally, we found out that conversion to Conjunctive Normal Form is hard and there are many edge cases to think about.

6 Conclusion and Future Work

The algorithms described in this report ultimately give the AI agent the ability to add new beliefs to its belief base, remove contradictory beliefs, and check whether a belief is entailed by the belief base. The AI agent, however, only works with propositional logic sentences. Future work on this AI agent

could include applying the belief revision to a specific purpose (such as Master Mind), implementing some natural language processing to represent a belief as something more meaningful than simply a combination of propositional logic symbols, or allowing the use of first-order logic to make the belief base representation more powerful. The algorithm that transforms propositional or first-order logic into CNF need to be modified to result in the correct CNF for all inputs.