Danmarks Tekniske Universitet

DTU

02180
Introduction to Artificial Intelligence

Assignment 2 - Logical Belief Revision Engine

Authors:
Connor Wall - s217170
Peter Revsbech - s183760
Kasper Solhøj Jørgensen - s194579
Tobias van Deurs Lundsgaard - s194616

Date: 02.05.2022

1 Introduction

This report will focus on the concept of belief bases and belief revision in propositional logic. We have developed a program which is capable of storing propositions in a belief base and updating it according to the important AGM rationality postulates. We will be explaining our understanding of the necessary concepts, as well as how we have implemented our solution. The implementations has been done in the Java programming language.

2 Design and Implementation of Belief Base

In this section we shall describe how we have implemented a belief base in our program. A belief base is a set of sentences which we currently believe to be true. The belief base should be able to receive new information and revise its previous beliefs according to the new observations, so the belief base is always consistent. We want to let the user input some propositional logic sentence and have it be stored and interpreted as individual logical operators and literals. There are several steps that needed to be taken in order to achieve this. First we need to be able to interpret strings as literals. Second we need to implement the logical operators that we know from the course, which can connect literals together or negate them. These are listed below along with the symbols they are represented by in this project:

Logical Operator	Representation in the Program
AND	&
OR	|
NOT	!
IMPLICATION	=>
BI-IMPLICATION	<=>

The logical operators and literals have been implemented as individual classes using an inheritance hierarchy. All the operator and literal classes inherit from the Propositon class directly or indirectly. So a proposition may be a connective of zero, one or two other propositions, depending on if it was a literal, a NOT or a binary connective respectively.

The belief base is stored in its own class, which has a list of propositions representing the current beliefs. The user may input a legal proposition, with which the belief base will be revised, so this class handles the revision and contraction algorithms. These will be explained further in section 4.1 and 4.2.

The contraction algorithm uses the DPLL algorithm, which is implemented in the class SAT, to check the satisfiability of a proposition. The implementation of the DPLL algorithm will be explained in section 3.1.

3 Design and Implementation for Checking Logical Entailment

In our revision agent, we will need to be able to check logical entailment. This means, that we can determine if a given proposition, ϕ, can be inferred from the knowledge base or not.

3.1 The DPLL satisfiability algorithm

To check logical entailment, we use a proof by contraction strategy. We negate the new information, ϕ, add it to the knowledge base, and see if this is now satisfiable. To check satisfiability we have implemented the DPLL algorithm, which is a backtracking-based search algorithm that determines the satisfiability of propositions on CNF form.

Initially, the algorithm creates a model where no value of any symbol is known. The overall idea is then the same as in the truth-table approach for checking satisfiablity, i.e. checking all combinations of truth values for all symbols. In each iteration, we assume both truth values of one new symbol, and evaluate each clause. The improvements over the truth-table approach then fall intro three categories: early termination and two types of simplifcations that are performed in each iteration.

Early termination means that when we try to evaluate the truth value of the binary operators (&,|,<=>,=>), we only evaluate one of the arguments, if that yields the answer. For example, when evaluating X|Y and we know that X=1, we don't need to evaluate Y.

Pure symbol simplification The algorithm looks for if any given symbol in the model is only represented with the same sign in all clauses (i.e. only positively or only negatively). If so, it is deemed a pure symbol, and its value is assigned so that its clause is evaluated to be true when checked. Then the algorithm can ignore clauses with this symbol because they are already known to be true.

The unit clause simplification finds any eventual unit clauses, and assigns their symbol the value needed to evaluate them to true. A unit clause is a clause consisting of only one (possibly negated) literal, and therefore there is only one way to satisfy it (i.e. X should have X=1 and !X should have X=0). In this case, we can simply assign the value that makes the clause true in the model. This will satisfy the unit clause, but may make other clauses false. This is however not a problem in checking satisfiability, since we are only interested in knowing if all clauses can be satisfied or not.

3.2 Time Analysis

In the worst case, this algorithm tries all possible combinations of all symbols. So if we have n symbols in our proposition, we will need to evaluate it 2^n times, as we would in the truth-table approach. In a lot of cases however, the DPLL

algorithm will terminate much quicker than this, because we don't need to try all combinations.

It is difficult to benchmark how much faster the algorithm is than the truth-table approach, since it depends completely on the given input. A worst case input would be one proposition that is very long and complex with no unit clauses, no pure symbols and no option for early termination. For this worst case input, their running times might be almost the same, whereas for some other inputs, the DPLL algorithm would finish in constant time and the truth-table would finish in exponential time. Since the result of a comparison on the algorithms is so dependant on the input, it does not seem meaningful to do.

3.3 Future Improvements

If there had been more time for the project, we could have made "propostion generator", that could generate a large number of different propositions of different lengths and complexity. This could be used to get a better idea of the performance differences of the DPLL alogorithm vs. a truth-table approach.

It would also be meaningful to add a max-time-parameter which sets a upper boundary for how long the DPLL algorithm can run. This upper bound would avoid the algorithm from running for an unreasonably long time, which is a real possibility with exponential time algorithms.

4 Implementation of Revision of Belief Base

4.1 Revision

Revision is a key part of updating a belief base. The purpose of belief revision is to ensure that when we add sentences to our belief base, we do not end up with a belief base containing contradictions. This can be defined mathematically with the Levi Identity which is as follows: $B * \phi := (B \div \neg\phi) + \phi$

The revision can be described in two parts. First we do contraction on the belief base with the negation of our belief. What this means is that we remove any elements in the belief base that contradict our new belief. After the contraction, we add the belief. This is exactly how we have done it in our code as well. The biggest part of the computation is the contraction.

4.2 Contraction Algorithm

Contracting a belief from a belief base first requires that we check if all current beliefs entail the new belief (see section 3 for entailment). If they do not, we do not need to remove any elements from the belief base. If they do entail the belief, we need to remove some of the existing beliefs according to some meaningful strategy. We need a priority system in our program that handles which beliefs should be more or less likely to be removed. Our priority system works as follows:

1. The highest priority is to delete as few beliefs as possible

2. The next priority is that the sum of the **age** of the deleted beliefs should be minimized.

In our program, the **age** of a belief is determined by the index upon which it is stored in our belief base list. The lower the number, the older the belief is. An example would be that we have our belief base containing the simple literal beliefs: (A, B, C, D, E). They have the ages: (0, 1, 2, 3, 4) respectively. We now want to add the belief ((!A & !E) | (!B & !C)). This new belief requires that we remove either A and E or B and C. The age of A + E is 4, and the age of B + C is 3, thus B and C will be removed.

Our algorithm for checking which beliefs should be removed is very thorough and will only ever remove the minimal number of necessary beliefs. However, it is potentially very slow. The way it finds the optimal set of beliefs to remove, is by trying all combinations of k propositions for all $k = 1, 2, ..., n$, where n is the number of propositions in the belief base. The number of such combinations for a given n is $\binom{N}{k}$. It will terminate early, when it finds a solution, but in the worst case the runtime of this algorithm is exponential in the number of propositions in the belief base.

We chose to use it anyway, since we valued the correctness of the algorithm higher than the time complexity in this case. See section 5.2 for a more in depth explanation.

One might also ask why deleting as few beliefs as possible has the highest priority, when there could easily be redundant information in the belief set, when a number of beliefs are equivalent. Our idea is, that many equivalent beliefs indicate a higher certainty that the beliefs are true.

4.3 AGM Rationality Postulates

The contraction and revision algorithms implemented in the engine live up to all of the AGM rationality postulates requested in the assignment description. We will now give a short argument why this is for each of them.

Contraction AGM postulates

1. Success: If ϕ is removed from the belief base, we can not then infer it from the belief base unless it is a tautology. This holds for our engine, which deletes enough propositions that ϕ is not a logical consequence of the belief base on contraction.

2. Inclusion: On contraction, we only delete and never add propositions to the belief base. So the result is a subset of the original belief base.

3. Vacuity: Removing unrelated propositions which are not tautologies has no effect on the belief base. Removing tautologies from the belief base is not allowed and will cause an error.

4. Extensionality: If ϕ <=> ψ then, removing one of them is the same as removing the other. This is true, since the propositions are logical consequences of the exact same subsets of the knowledge base, when they are logically equivalent. The contraction algorithm will therefore find the same subset of propositions to remove in both cases.

Revision AGM postulates

1. Success: This holds, since revising with ϕ adds ϕ to the belief base in all cases.
2. Inclusion: This holds, since revising with ϕ adds ϕ and may delete other propositions, but not add any. So the resulting belief base will be a subset of the original joined with ϕ.
3. Vacuity: This holds, since we only contract on revision, if ϕ introduces a contradiction to the belief base. So if the negation of ϕ can't be inferred from the belief base, then ϕ is simply added.
4. Consistency: This holds, since the algorithm will delete enough propositions from the belief base to make it consistent, if adding ϕ would otherwise introduce a contradiction. If one attempts to revise where ϕ is in itself a contradiction (e.g A !A), it will give an error and not be added.
5. Extensionality: This rule applies to belief sets and not belief bases. It still holds in the sense, that revising with ϕ and ψ will result in two belief bases that are logically equivalent, meaning that their corresponding belief sets will be equal.

5 Reflection

Working on the project has given us a much better understanding of belief revision and the underlying concepts. It has been interesting to challenge our understanding with the practical task of creating the engine.

5.1 Logical entailment and satisfiability

Checking logical entailment using satisfiability is a completely central part of the engine, and it may seem discouraging, that an algorithm with exponential time complexity is used for this. However, since the satisfiability problem is NP-complete, we know that no polynomial time algorithm solves the problem, and in practice, the DPLL algorithm seems to do a good job.

5.2 Contraction Prioritizing

As mentioned in section 4.2, our contraction algorithm always removes as few beliefs as possible. It will never remove a belief that could have stayed without contradicting the new belief. As mentioned, this makes it exponentially slower as more beliefs are added. We could think of a scenario in which we would want to sacrifice some precision in order to increase the speed of the program.

If our program was to be used by a robot meant to interact with people, we would want it to be faster so as to not halt the conversation. In such a scenario it would be okay to remove a couple of valid beliefs in order for the robot to respond to a human faster. It would be akin to how humans forget things during conversation as well.

We thought of a couple of ways to achieve this speedup. Firstly we could use a different algorithm. A faster algorithm would be one in which we keep removing beliefs from one end of our belief base to the other until we have no contradictions, and then add them back in the same order as long as possible without introducing a contradiction. This way we might remove some beliefs that had nothing to do with the contradiction, but finding the set to remove would take linear time in the number of beliefs. We also considered a combination of our existing algorithm and this new version. This would involve using our current algorithm for a set amount of time, and if this time limit is reached, we switch to the faster algorithm.

6 Conclusion

When making a belief revision agent Java, it is important to utilize the object oriented structure of the language to its fullest. By creating objects for each of our logical operators, we can effectively pattern match. Afterwards it is as simple as describing what patterns evaluate to certain truth values. In our code, we implemented a belief base as simple list of legal propositions from which we could do revision with our matching system. We wanted our revision to be as optimal as possible, and only delete as few beliefs as possible while preferring to delete the oldest beliefs we have over the newer ones. This has come at the cost of the program being fairly slow for larger belief bases. This slowdown comes from our resolution based implementation of logical entailment. When checking for logical entailment we have tried to optimize the speed as much as possible. To do this, we implemented the DPLL algorithm seen in the course. Initially we implemented it without handling pure symbol simplification or unit clause simplification, but these were both added for the sake of efficiency in the final version. The final version of our belief revision agent is capable of taking user input in the form of legal logical proposition in a terminal, and outputting the resulting revised belief base.