Belief Revision using Action Language A

Tyler Dwyer - 301 024 069

Dec 18, 2009


Introduction

It is commonly believed that we all have some sort of knowledge. The knowledge that you hold is revised when new information is received and it conflicts what you currently hold to be true. Whether it is a revelation about a complex matter such as finding out about being adopted \[8\], or something smaller such as the swan you caught is not actually white \[2][3\], there must be some sort of changes to your beliefs. These changes are problematic, questions arise like what should be changed, and how should it be changed. This report aims to implement a method to deal with these issues in a developing area of action languages.

Action languages are formula implementations of causality, meaning that one event can cause something to become true (or false) if it meets some preconditions\[4\]. These languages are useful in any agent which needs to reason about performing, or understanding actions in its world. Adding belief revision to these action languages will allow the agent to change its beliefs about its own initial state. Furthermore using a topological revision operator\[5\] no ‘user input’ plausibility ranking will be required for the belief revision.

This report will be organized into four main sections. The first section will be a brief overview of the field and current research. The second section will demonstrate a high level overview of the algorithm, and implementation. The third section will include test cases. The final section will be a short conclusion. An appendix will also be attached with instructions of how to compile and run the program.

1 Background

To best understand the field covered in this report, this section has been divided into two sub-parts. The first part will cover previous work done in the area of belief revision. The second part will cover action languages.

1.1 Belief Revision

To begin with belief revision as discussed in \[1][2\] and \[3\] we must first have some beliefs in our knowledge base. An example knowledge base is the swan example\[3][2\]. It is as follows:

- The bird caught in a trap is a swan
- The bird caught is from Sweden
- Sweden is a part of Europe
- All European swans are white

This can be compressed into logical statements such as\[3\]:

\[\begin{align*}
\text{isSwan} \\
\text{fromSweden} \\
\text{fromSweden} \Rightarrow \text{fromEurope} \\
\text{fromEurope} \land \text{isSwan} \Rightarrow \text{isWhite}
\end{align*}\]

From here we can conclude isWhite. However if we are given a new piece of information
such as \(-isWhite\), and try to add it to our knowledge base, we arrive at a contradiction with what we currently believe. This is where belief revision comes in. Something needs to be removed from our knowledge base to make it consistent, however it is unknown what we should remove. An obvious (and ridiculous) answer would be to remove everything from the knowledge base so the only thing that is known is \(-isWhite\). To further illustrate that this is not a rational coarse of action a few guidelines have been developed to aid with belief revision. The AGM\(^2\) posulates do just that, and are explained below\([8]\):

(K*1) For any sentence \(A\) and any belief set \(K\), \(K * A\) is a belief set.

The first postulate, called the closure postulate simply states that the output of the revision function must be a belief set.

(K*2) \(A \in K * A\)

The second postulate, called the success postulate, states that the new information has “absolute priority over the beliefs originally entertained”\([3]\). This means that the new information is always true, and always fully accepted.

(K*3) \(K * A \subseteq K + A\)

(K*4) If \(\neg A \notin K\), then \(K + A \subseteq K * A\)

The third and fourth postulates, called the expansion postulates cover the cases where the new information does not generate a contradiction with the knowledge base. In this case the new information is simply added, and nothing is removed.

(K*5) \(K * A = K \bot\) only if \(\vdash \neg A\)

The fifth postulate, called the consistency preservation postulate, ensures that if A is logically possible, then the result of \(K * A\) should be consistent.

(K*6) If \(\vdash A \leftrightarrow B\), then \(K * A = K * B\)

The sixth postulate, called the extensional postulate, states that logically equivalent sentences should result in the same revisions.

(K*7) \(K * A \land B \subseteq (K * A) + B\)

(K*8) If \(\neg B \notin K * A\), then \((K * A) + B \subseteq K * A \land B\)

The seventh and eighth postulates, called the conjunction postulates, state that if we revise by A and B, this should be a subset of revising by one, then expanding by the other (given that the other does not contradict)

From these eight postulates we can start to build a revision operator, however a new problem quickly arises. In the case where our new information contradicts with our knowledge base we need to know how to remove the contradiction. This will involve removing some information from our knowledge base, however choosing the ‘most’ appropriate term or sentence to remove is, at this time, unspecified.

\(^1\)where \(\neg\) represents classical negation

\(^2\)AGM stands for the 3 creators, Alchourrón, Gärdenfors and Makinson
To solve this problem, an entrenchment, or plausibility ordering on the information can be used to determine what to remove. This ordering allows the revision operator to pick the least entrenched/plausible information to remove that allows us to correct the contradiction.

One implementation of this is to directly order the sentences in the knowledge base. This gives the knowledge base an epistemic entrenchment[8]. Now when we receive a new piece of information that generates a contradiction, we can simply determine which sentences in the knowledge base generate the contradiction, and remove the sentence (or sentences) that are the least entrenched.

Another method of implementation is to order the information not on the sentences, but on the worlds which the knowledge base represents. These worlds contain the set of possible values for each term for the knowledge base. For example with our swan example, the only possible world from our knowledge base is:

\[
\text{isSwan, fromSweden, fromEurope, isWhite}
\]

However this is only the world that is entailed from the knowledge base, other worlds exist however they are less plausible such as:

\[
\text{isSwan, ¬fromSweden, ¬fromEurope, isWhite}
\]

Which states that it is a swan, and it is white, however it is not from Europe (also not from Sweden). Now given these two possible worlds we can order them accordingly. This generates a System of Spheres which was outlined by Harper [1977], Grove [1988], Morreau [1992] and others[8].

With this system of spheres implementation we can simply remove all worlds which contradict with our revision term, and take the most plausible world that is left over. This is the method implemented, and will be discussed at length later on.

1.2 Action Languages

Action languages is discussed in [4][7][6] and [5] as a formal implementation of causality into logic. Given an action A, we can define its effects based on preconditions. Action languages use a transition system which takes a state \( S \), applies action \( A \) to it, resulting in a new state \( S' \). A transition system can be represented, or thought of as a labeled directed graph. Each vertex is a situation, and each arc represents an action leading to a resulting state.

There are several different action language that have been developed, however in this implementation only action language 'A' will be investigated. Action language 'A' was the first, and the simplest. It only includes two operators, \( \text{initially} \) and \( \text{causes} \). The \( \text{initially} \) operator sets the starting values of a fluent. The \( \text{causes} \) operator specifies the action in the form \( A \text{ causes } L \text{ if } F \) where \( A \) is the action name, \( L \) is a literal which will be affected by the action, and \( F \) is a conjunction of literals which represent the precondition to the action. An example of an action language 'A' knowledge base is shown below\(^3\)[7]

---

\(^3\)This example was originally in the context of action language \( A_k \)
initially !burnOut
initially !bulbFixed
changeBulb causes burnOut if switchOn
changeBulb causes bulbFixed if !switchOn
turnSwitch causes switchOn if !switchOn
turnSwitch causes !switchOn if switchOn

2 Algorithm and Implementation

To best explain the methods used, and the implementation created, this section is divided into three sub-parts. The first part will outline belief revision, how it has been implemented, the methods used, and brief demonstrations. The section part will outline action languages, its implementation, methods used and some brief demonstrations. The final section will outline how both ideas have been merged to allow for belief revision to occur within an action language. The implementation, methods used and brief demonstrations will also be shown.

2.1 Belief Revision

To perform belief revision several steps must be taken, they are outlined here:

2.1.1 Step 1 - Adding Knowledge

The first step is to load information into the knowledge base. This information is what our agent believes to be true before any revision happens. The example that will be outlined here is the swan example, that was looked at in the background section. In this implementation the following propositional operators are been implemented: $\land$, $\lor$, and $\rightarrow$ and represented as ‘(and p q)’, ‘(or p q)’, and ‘(imp p q)’ where p and q are boolean terms. The modified version of the knowledge base is shown below:

```plaintext
isSwan
fromSweden
(imp fromSweden fromEurope)
(imp (and fromEurope isSwan) isWhite)
```

In the implementation to add knowledge into the program there are two options, either by selecting a pre-made example, or by manually entering in the sentences of your own knowledge base. The pre-made examples consist of a P Q example, which demonstrate the different operators use, and the swan example used in this report. To load the swan example simple select "White Swan (Propositional Logic)" radio button. This loads the program with the knowledge base, and sets it to work with propositional logic, which is shown in Figure 1.
2.1.2 Step 2 - World Generation

The next step is to generate all the possible worlds for the language of the knowledge base. This is done by scanning through the knowledge base for all the terms used (negations are removed if necessary). The language for the swan example is "isSwan, fromSweden, fromEurope, isWhite". From this language we can generate all the possible worlds, note that there are $2^N$ possible worlds where N is the number of terms in the language. In this case N is 4, so we will generate 16 possible worlds. The possible worlds will range over all possible values of worlds, as shown in figure 2. Note figure 1 also shows the generated worlds, however the generation of worlds is implemented in the next step.
GroveRevision(revisionTerm T, worldList Worlds)
    for each world in Worlds
        if world contains !T
            delete world
    while ( Level-0 contains zero worlds and Worlds.count > 0)
        move each world down one level

Figure 3: Pseudo-code for Grove Revision

2.1.3 Step 3 - System Of Spheres

The next step is to generate an ordering of plausibility on all the worlds generated in the previous step. These worlds are ordered from most plausible at level 0, and less plausible worlds at higher levels, this is the system of spheres. To generate this ordering we can start by setting the level-0 worlds as the worlds the agent currently believes in. To find these, each world in we generated in step 2 is tested against the agents knowledge base, if it does not make a contradiction (holds true), then it is put at level 0. The other worlds which were not entailed by the knowledge base are randomly assigned to levels 1, 2 or 3. They are randomly assigned plausibility levels (other than level-0 worlds) because the plausibility of a world is problem specific and defined by the user. Automatic systems can be implemented here for generating orderings, such as the hamming distance between worlds. Using the hamming distance however may not provide intuitive results, for example in the swan problem the following sentence:

    fromSweden, !fromEurope

Which implies that Sweden is not apart of Europe, however the bird is still from of Sweden would be considered more plausible than:

    !fromSweden, !fromEurope

Which implies that the bird is simply not from Europe, and in turn, not from Sweden.

Both steps 2 and 3 can been executed by pressing the ”Find Worlds” button. This will display the system of spheres generated in the center text box on the screen as shown in Figure 1. Note that the plausibilities orderings can be increased or decreased by first selecting the worlds that you wish to change and using the ”increase level” and ”decrease level” buttons located below the center text box.

2.1.4 Step 4 - Revision

The first method of revision that is discussed is Grove revision [reference]. This revision method has been implemented by taking a revision term \( \phi \) and checking it against each of the possible worlds. If the world contains \( \neg \phi \) then the world is deleted, if it contains \( \phi \) it is left as is. Then if level-0 does not contain any worlds after all the \( \neg \phi \) worlds have been removed, all the worlds are moved down one level until level-0 contains at least one world (or no worlds are left). The pseudo-code is shown in figure 3. This revision can be performed
by entering a $\phi$ term in the "Revise by" box, then pressing the "Grove Revise" button. The revised system will appear in the center box, shown in figure 4. Note that in this implementation all levels have been included in the output of Grove revision. This has been done for understandability, it does not represent a new system of spheres of the revised system. For a further demonstration that Grove revision does not work iteratively, revising by $\phi$ then by $\neg\phi$ will always result in all worlds being removed.

The second method of revision implemented deals with the issues of iterative revision.

![Figure 4: Worlds after Grove revision with the term !isWhite](image)

The difference between Grove revision and Spohn revision is that instead of deleting the $\neg\phi$ worlds, the are moved down into lower plausibility levels (levels with higher values), and the $\phi$ worlds are moved down (until at least one is at level 0). A variable needs to be added to this implementation to control how far to move down $\neg\phi$ worlds, it has been called 'D'. The pseudo-code for this algorithm is shown in figure 5. Spohn revision can just performed just as Grove revision was, except by using the 'Spohn Revise' button. Note that the value of D is initially set to 1, however can be changed by using the '+' and '-' buttons next to the 'D =' box. Figure 6 demonstrates Spohn revision with !isWhite as a revision term. Furthermore, when Spohn revision has been performed a new system of spheres will shown in the center box, note that if one revises by $\phi$ then $\neg\phi$ the resulting system of spheres will be the original world ordering, which is what you would expect.
SpohnRevision(revisionTerm T, worldList Worlds)
for each world in Worlds
    if world contains !T
        move world down D levels
    while (Level-0 contains zero worlds)
        move T worlds down one level

Figure 5: Pseudo-code for Spohn Revision

2.2 Action Languages

This section on action languages is, like belief revision, best explained in several steps. We will outline the steps to add action language knowledge to a knowledge base, from this knowledge generate an initial state, perform an action, then perform recursive actions.

2.2.1 Step 1 - Adding knowledge

As in belief revision, the first step in implementing action languages is to add knowledge to an agents knowledge base. The example that will be used here will be the light bulb fixing robot from the background section. Note that the operators have been modified to make parsing easier, the modified knowledge base is shown below:

(initially !burnOut)
(initially !bulbFixed)
(causes changeBulb burnOut switchOn)
(causes changeBulb bulbFixed !switchOn)
(causes turnSwitch switchOn !switchOn)
(causes turnSwitch !switchOn switchOn)

Note that the cause operator is organized as follows (causesABC), where A is the action name, such as 'change bulb'. B is the result of performing the action, such as 'switchOn', and C is the precondition that must be met before this action has any affect.

To load in this knowledge base into the program select the radio button 'Light Bulb Robot (Action Lang)', as shown in Figure 7. Terms can be added and removed from the knowledge base just as was done with belief revision.

2.2.2 Step 2 - Generating an initial state

The next step is to generate an initial state that the agent believes it is in. This state is done by looking through the knowledge base and finding the language that is used in it. This language includes preconditions, action effects, and initial clause terms from the knowledge base, however does not include actions names (such as changeBulb). From this language all possible worlds are generated from it (as was done in belief revision), then the initial clauses
are compared to each world. Any worlds that does not contain all the *initial* clause from the knowledge base are removed. The result from this is all the possible worlds that the agent could possibly be in with the *initial* clause terms as constraints. The initial worlds, also called the initial state is shown in Figure 7. This can be produced by pressing the ‘Find Worlds’ button.

### 2.2.3 Step 3 - Perform actions

To perform an action we now have to implement a transition system. This transition will take the agent from one state (containing possibly many worlds) to a new state where the action has been applied. This has been implemented by transitioning between one world to another using a single action at a time. When an action is chosen to be applied, all the different action preconditions and effects are extracted from the knowledge base. There may be several sets of precondition/effect pairs for the same actions\(^4\), this is because the actions are non-deterministic and are based upon the preconditions. With this set of action precondition/effect pair, each world in our current state is tested against each precondition/effect pair. If the world contains the preconditions the action affects are applied to the world, meaning that the world is changed to contain the action effects. An example of this would be if the world contained \(\neg \phi\) and the action effect was \(\phi\) the world would be

\(^4\)Such as the the two precondition/effect pairs for changeBulb and turnSwitch in the light bulb robot example
changed to $\phi$, note that if the action effect was $\neg\phi$ nothing would be changed. The pseudo-code algorithm for the transition system is described in figure 8.

In this implementation the transition system and applying actions has implemented so it is easily accessible by the user. To apply an action to the current state (the one in the center text box), simple select an action from the "Actions" box, then press the "Apply selection action" button. The resulting state will be shown in the center text box. An example of this is apply the action `changeBulb` to the initial state of the light bulb robot example. The starting initial state is:

```plaintext
!burnOut,!bulbFix,switchOn
```

The resulting state is:

```plaintext
burnOut,!bulbFix,switchOn
!burnOut,bulbFix,!switchOn
```

As the above example illustrates the agent either burning out as the switch is one, or fixing the bulb if the switch is off. Figure 9 demonstrates this.

### 2.2.4 Step 4 - Performing recursive actions

The last step is to be able to perform multiple action languages on recursive states. An example of this in our light bulb example would be to first, perform the `turnSwitch` action, then apply the `changeBulb` action, on the new state. This can be done by simply apply both
[returnState] = Transition(initialState, action)
    actionName = action.actionName()
    setOfPreconditions = action.preconditions()
    setOfEffects = action.effects()
    for each world in initialState
        for each precondition in setOfPreconditions
            if (world contains precondition)
                newWorld = world.apply(effect)
            if (newWorld not set) newWorld = world
                add newWorld to newWorldState
    return newWorldState

Figure 8: Pseudo-code for a action language transition system

actions after one another. However this becomes slightly more complicated when you want to find all the possible worlds that would result after apply a number of actions (regarded to as depth). The light bulb example does not illustrate this point very well so a new example will be introduced. In the new example (called the box example) there is an agent, three rooms and a box. The agent is in the first room, the box is in the third room, the agent also has the abilities (actions) to move between rooms, and open the box. The knowledge base for this example is provided below:

(initially R1,!R2,!R3)
(initially !boxOpen)
(causes MoveF R2,!R1 R1)
(causes MoveF R3,!R2 R2)
(causes MoveB R1,!R2 R2)
(causes MoveB R2,!R3 R3)
(causes openBox boxOpen R3)

With this example we can new can see that after finding all the possible states after a number of actions have been executed is more complicated. To find all the possible worlds after a depth of actions in this implementation, simply set the depth box to how deep you would like to search\(^5\) then press the "Find All States" button. This will generate all the possible worlds for each depth, and will be shown in the center box. Figure 10 demonstrates the box example with depth = 2. Note that depth zero is the initial state as zero actions have been applied, also there are three depth 1 groups, one for each action in the knowledge base, MoveF, MoveB, and openBox. Under each depth 1 groups, it is again divided into three groups, one for each action. This subdivision of groups is due to the recursive nature of the algorithm used. The pseudo-code for the algorithm is shown in figure 11 for completeness.

\(^5\)The maximum depth recommended is 5, as higher may crash the program due to too many worlds generated
2.3 Belief Revision in Action Languages

At this point we have successfully implemented both belief revision using propositional logic, and action language A. The last step is to combine the two, to give us the ability to change an action language agents beliefs. This is done by using the recursive states shown in step 4 of action languages (section 2.2.4), to represent a system of spheres which can be used by the belief revision implementation. This method of using the transition system to create a system of spheres is called topological belief revision[5]. This is done by collapsing all the recursive worlds down to just their depth levels, and using the depth value as the level for system of spheres. This has a very strong assumption that the depth of actions, or in other words the number of actions performed, relates to the plausibility rating of the world.

To create a system of spheres ordering from a list of states generated in section 2.2.4, simply press the 'Convert to System of Spheres' button. This is shown in figure 12. From this ordering we can now perform Grove or Spohn revision on the world, just as in belief revision. A revision of boxOpen of the box example is shown in figure 13. This Revision shows that in the situation where the agent believes it is initially in R1 and the box is closed, then learns new information that the box is now open. From this system of spheres ordering the agent can reason that the most plausible world is that it is now in R3, and the box is open.

---

6This will require all the states to a certain depth to be created, to do this press Find Worlds, then Find all States
Figure 10: Apply many actions - Box Example - Depth = 2

\[
\text{findAllStates}(\text{actions}, \text{world}, d) \\
\text{for each action in actions} \\
\quad \text{newWorld} = \text{Transition}(\text{world}, \text{action}) \\
\text{if (d < depth)} \\
\quad \text{findAllStates}(\text{actions}, \text{newWorld}, d+1)
\]

Figure 11: Pseudo-code for finding all states for all actions

The agent knows this as it is the highest level in the system of spheres\(^7\). It is also possible in this implementation to find which actions must have been performed to get to this state as is seen in the 'Actions Performed' box in figure 13. To do this select a world, and press the 'Find' button in the lower right hand corner. Furthermore the implementation performs intuitively as it shows that it is more plausible that the agent performs only 3 actions to open the box, instead of 4 where the fourth action could be attempting to open the box in room 2 or any other redundant action.

\(^7\)Note with normal grove revision, as discussed in the belief revision section, the worlds would automatically be moved up to level 0 if it was empty. That has been disabled here to illustrate which worlds have been removed, and the depth of worlds remaining
3 Conclusion

In conclusion we have successfully implemented belief revision, using two different techniques. Then separately implemented action language A with the abilities of performing actions individually or recursively. Then integrating the two implementations to allow for belief revision in an action language. Furthermore the integration of the two provides reasonable and rational results that could allow an agent to reconstruct its beliefs about its initial state given some new information using a depth of actions to convert an list action language worlds to a system of spheres.

Appendix

Running the program

First Extract Project.tar
Double click on BeliefRevision.exe in the "Executable" folder
Follow the steps outlined in the Algorithm section

Note, if the BeliefRevision.exe does not work, it may require the Microsoft .NET 3.5 package.
Figure 13: Revision of boxOpen performed on box example with depth = 5

To Compile this program using Visual Studio 2008:

Open the file BeliefUpdate.sln, which is located in the 'BeliefUpdate' folder. This will open visual studio 2008, from here you can view the code, and compile it by pressing the green arrow button, or using the menu (Debug > StartDebugging), or by pushing F5. Once visual studios has opened, you may need to open all the code files using the 'Solution Explorer' which may be docked at the side of the screen.

The program contains three classes, the first is the Form where all the GUI buttons and text box code is stored. The second is the interpreter where much of the calculations for queries, actions and states is carried out. The third is the world class which stores the worlds, and does some reasoning about them.

References


