
An Attempt to Generalize AI

Part 3: Forgetting

By Paul Almond

13 March 2010

Website:

<http://www.paul-almond.com>

E-mail:

info@paul-almond.com

This is the third in a series of articles attempting to give an overview of how minds may work and how similar systems could be implemented in computers. The first article described a probabilistic, hierarchical modeling system intended to provide a general ontology. The second article described the use of this for planning actions. The hierarchy as described so far is idealized. One way in which it is impractical is that all pattern instances exist in the hierarchy permanently. It is implausible that the human brain works in this way. It would mean, for example, that every single input to a light sensitive cell on your retina was stored permanently, so that your brain would contain a detailed “video recording” of your entire life. A system working like this would waste computing resources on dealing with pattern instances that are no longer relevant. A basic forgetting procedure is described, in which a pattern instance is erased when it has a known state and no longer has any effect on the hierarchy: This is when it is no longer being used as a pattern input by any other pattern instances which do not yet have known states. The possible desirability of increasing or reducing the amount of forgetting is discussed, as well as ways of doing this. The basic forgetting procedure is also related to human experience of memory and forgetting. The two kinds of forgetting are not exactly the same, but we should expect some correspondence between them.

Table of Contents

1 Introduction	5
2 The Hierarchy	6
2.1 General Idea of the Hierarchy.....	6
2.2 An Example of Fixing of Pattern Instances in the Hierarchy.....	8
2.3 A Note on the Construction Specification	10
3 The Problem of Pattern Instances Persisting Permanently	11
4 Basic Forgetting.....	13
4.1 Simplifying Assumption.....	13
4.2 Uses of Pattern Instances	13
4.3 The Basic Forgetting Procedure.....	15
4.4 The General Effects of Forgetting on the Hierarchy	15
4.5 Basic Forgetting as a “Baseline” System	17
4.6 An Example of Basic Forgetting	17
5 More Sophisticated Forgetting	19
5.1 Why We May Want to Use <i>Less</i> Forgetting	19
5.2 Why We May Want to Use <i>More</i> Forgetting	19
5.3 Using <i>Less</i> Forgetting	20
5.4 Using <i>More</i> Forgetting	20
5.5 Still More Sophisticated Forgetting	21
6 Human Experience of Memory and Forgetting	22
7 Conclusion.....	24
8 Bibliography	26

Table of Figures

Figure 1: Patterns and the Hierarchy.....	7
Figure 2: An Example of Fixing of Pattern Instances in the Hierarchy	8
Figure 3: General Effects of Forgetting on the Hierarchy	16
Figure 4: An Example of Basic Forgetting	17

List of Abbreviations

- AI artificial intelligence
- EFS evaluation function score

1 Introduction

This article is the third in a series about artificial intelligence (AI) and how our own minds might work. The first article, *An Attempt to Generalize AI - Part 1: The Modeling System*, should be read before this one and is available at <http://www.paul-almond.com/AI01.pdf>.¹ The second article, *An Attempt to Generalize AI - Part 2: Planning and Actions*, is at <http://www.paul-almond.com/AI02.pdf>.² (Reading the second article before this one is suggested, but not absolutely necessary, as the focus here is on the hierarchy itself, rather than actions.)

Both of these articles together were intended to give an idea of how humans may model the world and plan actions.

The system as described so far is idealized. It is impractical in two obvious ways: All pattern instances are maintained permanently in the hierarchy and all pattern instances of a pattern are explicitly represented. In this article I will be showing how we deal with the first issue – pattern instances being stored permanently in the hierarchy. This will be done using a process of *forgetting*.

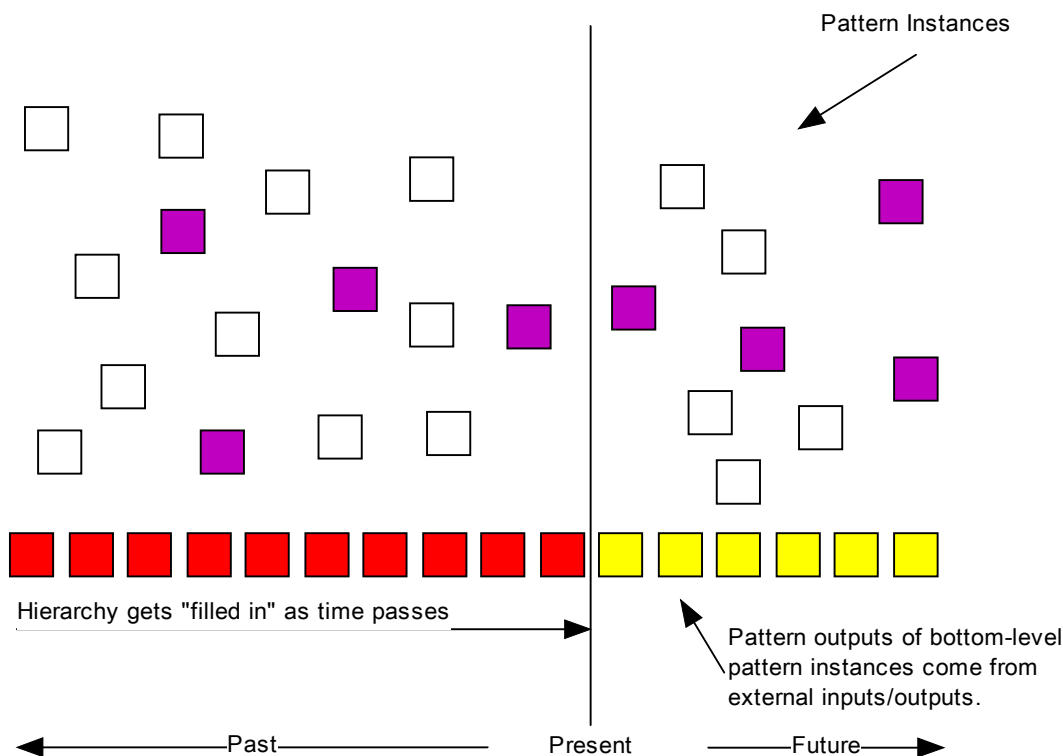
¹ Almond, P. (2010). An Attempt to Generalize AI - Part 1: The Modeling System. *paul-almond.com*. <http://www.paul-almond.com/AI01.pdf>. (Also available at <http://www.paul-almond.com/AI01.doc>.)

² Almond, P. (2010). An Attempt to Generalize AI - Part 2: Planning and Actions. *paul-almond.com*. <http://www.paul-almond.com/AI02.pdf>. (Also available at <http://www.paul-almond.com/AI02.doc>.)

2 The Hierarchy

2.1 General Idea of the Hierarchy

The system described in the previous articles is a hierarchy based on patterns. Each pattern is a set of pattern instances, distributed throughout the hierarchy and described by a pattern specification. The pattern specification consists of a logic specification, describing how each pattern instance of a pattern determines its pattern output from its labeled inputs, and a construction specification, describing how each pattern instance connects its labeled pattern inputs to the pattern outputs of other pattern instances. The bottom level of the hierarchy consists of special pattern instances, each corresponding to some external input/output event occurring at some instant. These pattern instances record the inputs/outputs that have occurred over a period of time. The rest of the hierarchy is built on top of this according to the pattern specifications of patterns. (See Figure 1, on page 7.)



Key

- Pattern instances of the same pattern. All these pattern instances are related in some way.
- Pattern instances of other patterns.
- Bottom-level pattern instances corresponding to external inputs/outputs that have already occurred.
- Bottom-level pattern instances corresponding to external inputs/outputs that have yet to occur.

Figure 1: Patterns and the Hierarchy

Because the pattern instances of a pattern are all related by being described by the same pattern specification, statistical observations of pattern instances with known pattern inputs can be used to assign probabilities to pattern instances with partially and probabilistically known pattern outputs. This can be used to propagate probabilities up the hierarchy to determine probabilities for high-level pattern instances, based on previous inputs/outputs, and down the hierarchy, to fill in the parts of the hierarchy corresponding to future inputs/outputs.

The hierarchy can be considered to be a collection of probability values – pattern instances – each of which collapses at some point into a known state, with this collapse process being ongoing as inputs and outputs occur. When a pattern instance has collapsed – when its state is determined, directly or indirectly, by inputs or outputs that have already occurred – it is *fixed*. The fixed pattern instance is then used to provide statistical data for the pattern to which it belongs, and it also serves as a pattern input for any other pattern instances that use it.

All this enables the hierarchy to predict future inputs/outputs probabilistically, based on previous inputs/outputs and this principle is used in selecting outputs.

2.2 An Example of Fixing of Pattern Instances in the Hierarchy

Before I go into the issue of forgetting I will give an example of fixing of pattern instances. Figure 2, below, shows an example portion of the hierarchy and the following discussion will relate to it.

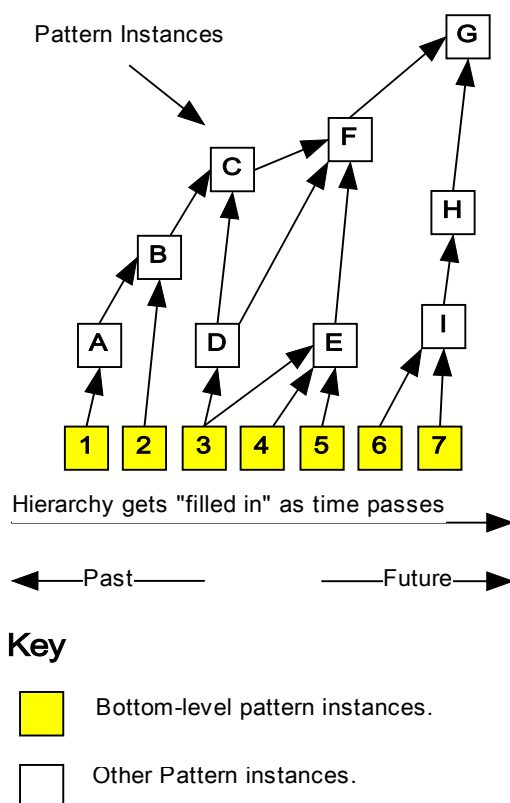


Figure 2: An Example of Fixing of Pattern Instances in the Hierarchy

We will start by assuming that the input/event corresponding to Pattern Instance 1 is about to occur. At this stage, all the pattern instances in the diagram have only probabilities, assigned by statistics application (normally based on probabilistic or partial knowledge of their inputs and the statistics of previous patterns, with known pattern inputs, that were fixed).

When the input/output corresponding to Pattern Instance 1 occurs, Pattern Instance 1 is fixed, taking the value of this input/output. Pattern Instance A, because of its logic specification, depends only on Pattern Instance 1, so Pattern Instance A is fixed.

When the input/output corresponding to Pattern Instance 2 occurs, 2 is fixed. B depends only on A and 2, which are both now fixed, so B is fixed.

When the input/output corresponding to Pattern Instance 3 occurs, 3 is fixed. This in turn causes D to be fixed. C is dependent only on B and D, which are both now fixed, so C is fixed.

When the input/output corresponding to Pattern Instance 4 occurs, 4 is fixed.

When the input/output corresponding to Pattern Instance 5 occurs, 5 is fixed. E is dependent only on 3, 4 and 5, all now fixed, so E is fixed. F is dependent only on C, D and E, all of which are now fixed, so F is fixed.

As pattern instances are fixed, it tells us something about the pattern inputs of the ones that are not yet fixed, allowing the statistic information we have to be used to assign them probabilities. These probabilities tell us still more about the inputs of other pattern instances which use them, allowing them to be assigned probabilities. The statistical information for this all comes from analysis of the fixed pattern instances. Each pattern instance belongs to a pattern, and by counting the frequency with which different combinations of inputs occur when pattern instances of a particular pattern are fixed, we can get a probability for a pattern instance of that pattern having a particular pattern output when we only have partial or probabilistic knowledge of its pattern inputs. In this way, probabilities are propagated through the part of the hierarchy corresponding to the future.

Later, I will use this diagram again, as an example of how forgetting can work.

One point I will make here is that the statistical aspect of the system is based on the idea of groups of pattern instances belonging to a pattern, and for a very general ontology, the relationship between the pattern instances in a pattern should not be restrictive, as would be the case if some kind of simple, geometrical analogy were used. It should be possible to construct a pattern based on *any formally describable relationship* connecting a group of pattern instances.

2.3 A Note on the Construction Specification

In the previous articles, I have described patterns as *constructive machines* – each pattern using a construction specification to direct the wiring of its set of pattern instances into the hierarchy. This is not essential. The important feature of a pattern is that its pattern instances need to have something in common about the way they are wired into the hierarchy – and it must be possible to specify this “something” in a very general way.

One way of achieving this is having patterns using construction specifications to connect their pattern instances, as has been described, but an alternative is to do things the other way round, with large numbers of pattern instances being regularly connected up to the hierarchy, perhaps randomly, with each pattern construction specification examining these pattern instances, and how they are connected up, and determining which meet the requirements for inclusion in the pattern.

If things are done this way, inclusion in a pattern might not be all or nothing: It could have a “fuzzy” aspect, in which a pattern instance could be included to varying degrees in one or more patterns, depending on how well it matches what the “construction” specifications of these patterns are looking for. If pattern instances can have varying degrees of membership of patterns, it would mean that the statistics for patterns would be weighted appropriately. If a fixed pattern instance is only weakly a member of a pattern, then it will have a minimal effect on that pattern’s statistics. If a pattern instance, which is still unfixed, is only weakly a member of a pattern then the statistics application process will mean that that pattern’s statistics have only a minimal effect on the probability for that pattern instance.

3 The Problem of Pattern Instances Persisting Permanently

The system as described so far keeps pattern instances in the hierarchy permanently. In Figure 1, on page 7, this would mean that the number of red pattern instances gets progressively larger, without limit, as pattern instances are fixed with values as they become known and remain in the hierarchy. Every input/output occurring at some instant is stored permanently, and will still be there years later, and so will any pattern instances derived from these. We might want *some* information to stay in the hierarchy permanently, but this means *everything*. For example, if the system has a video camera as an input device, it means that every time a pattern instance is used to encode a captured pixel, that pixel is stored permanently.

Most people who have really thought about the issue would be skeptical about any suggestion that humans work in this way. For example, if I suggested that every single input from a light sensitive cell on your retina is stored permanently, so that your brain somehow contains a high resolution “video recording” of every detail of your life so far, this should sound implausible. If you saw a vase ten years ago, and it played some important role in your life, then that abstraction of the “pixels” in the images from your eyes might be of interest a long time later, but one of the individual low-level elements making up an image of the vase at some moment would be of no interest, in itself, years later: The brain clearly deals in abstractions.

Similarly, in the hierarchy, we should expect the specifics of such low level information to be of much interest for only a short time. After that, we should only be interested in any statistical information we have gained from the pattern instance, and any higher-level pattern instances which have used it as an input to generate their own pattern outputs.

Some readers may wonder why we should do this. Every year we seem able to get more computing power for the same money, so why not just assume that future computers can deal with this and let the pattern instances accumulate without limit? The reason is that, no matter how much computing power we have, we could be using it for something better. The system as currently described explicitly represents all pattern instances of a pattern, but I will show later how the system can be made to explicitly represent only some pattern instances of a pattern. How many pattern instances we can have in the hierarchy depends on how much computing power we have. The available computing power limits the number of patterns we can have and/or the number of pattern instances we can explicitly represent for each pattern. If we are using a lot of the computing capacity for storing low-level pattern instances that are no longer of interest, we are reducing the computing capacity we have available for use in storing pattern instances that *are* of interest.

We therefore need to ensure that pattern instances are not stored in the hierarchy for too long. This applies to very low-level pattern instances, such as those on the bottom level of the hierarchy, but there is no sudden cutoff point between a “low-level” pattern instances and a “high-level” pattern instance. In general, the more abstracted a pattern instance is – the further away it is from the bottom level of the hierarchy – the longer we should be keeping it. I will now describe a way of doing this. Some readers may have already seen how this is going to work. In fact, from what I have said here, I am *hoping* it may seem obvious, because it is intended that the hierarchy described in the first article should make this the case.

4 Basic Forgetting

4.1 Simplifying Assumption

I will define a basic method of removing pattern instances from the hierarchy – a *basic forgetting procedure*. We may want the behavior of the system to differ from this: We may want it to do more or less forgetting than such a basic procedure, but the basic procedure will serve as a kind of “baseline” for forgetting: the forgetting which will occur in the absence of any additional, more complex intervention.

For now, I will make the simplifying assumption that no new patterns will be added to the hierarchy, because this makes the uses of pattern instances reasonably limited.

4.2 Uses of Pattern Instances

The issue we are looking at here is when we can remove a given pattern instance from the hierarchy: When do we no longer need it? To answer this we need to look at what we need pattern instances for, so we can decide when that need has been fulfilled.

Each bottom-level pattern instance corresponds to a single input/output occurring at some instant. This means that that the value of a pattern instance does not change as different input/output values occur. When the pattern instance corresponds to an input/output that still has to occur in the future, its value is only known probabilistically. When the input/output corresponding to a pattern instance actually occurs, it is assigned the value of that input/output as its pattern output value and keeps this value permanently.

Other pattern instances, above the bottom level, have pattern output values dependent on the pattern outputs of other pattern instances. When the pattern instances serving as inputs to such a pattern instance have not yet all been assigned values, its pattern output value is only known probabilistically. When all the inputs for a pattern instance are known, it is assigned a value, which it keeps permanently.

In other words, pattern instances are not like neurons or logic gates in this respect. They do not switch from state to state dependent on what is going on in the system. A pattern instance starts off as a probability, which may change over time, because its state depends on what is going to happen with inputs/outputs in the future, but at some point its state becomes known and is “fixed” permanently.

I will ignore the case of a pattern instance which is dependent, directly or indirectly, on future inputs/outputs – one with only a probability – and instead consider only the case of a pattern instance which has been “fixed” – which has been assigned a pattern output value due to the values of all its inputs (either external inputs/outputs or other pattern instances) being known: Every pattern instance is in this situation eventually.

A “fixed” pattern instance has the following purposes:

- **Providing statistical data** – It can be used to make a contribution to the statistical data for the pattern to which it belongs. That data gives an indication of the frequency with which each combination of pattern inputs occurs for the pattern instances of that pattern.
- **Serving as inputs to other pattern instances** – Once a pattern instance has a known pattern output value, that value can be provided as a pattern input to any pattern instances that are connected it, so that such pattern instances, using any other pattern instances as well, can set their own pattern outputs.

Beyond this, a pattern instance has no purpose, so for how long is a fixed pattern instance useful? The answer is: probably not very long.

Considering, first, the use of the pattern instance in providing statistical data, this is available as soon as the pattern instance is fixed. Once we know all of the pattern instance’s input values, we can immediately update the records indicating the frequencies with which different combinations of inputs occur for that pattern appropriately. After that, we no longer need the pattern instance for this purpose. We can assume that this condition is met as soon as a pattern instance is fixed, so we can effectively ignore this condition.

Considering the use of a pattern instance to serve as inputs to other pattern instances, the pattern instance is only useful for this as long as at least one other pattern instance exists which uses it as a pattern input and has not been “fixed” yet. That is to say, it is useful as long as at least one other pattern instance exists which uses it as an input and has other inputs/outputs that are not yet known. As soon as all the pattern instances which use a particular pattern instance as inputs have been “fixed”, that pattern instance is no longer needed. This means that the time for which we have to keep a pattern instance is dependent on other pattern instances. If a pattern instance has been fixed, and is needed by another pattern instance as a pattern input, and this other pattern instance has other pattern inputs which do not become known (except probabilistically) for a long time, then that pattern instance must remain in the hierarchy. On the other hand, if a pattern instance has been fixed and no other pattern instances, the states of which have yet to be resolved, need it as an input, the pattern instance can be discarded. One pattern instance is therefore able to “hold onto” another pattern instance – to force it to continue to exist – by requiring it as an input, while waiting for the values of its other inputs to become known.

4.3 The Basic Forgetting Procedure

Based on the above, the basic forgetting procedure is as follows.

Basic Forgetting Procedure

When a pattern instance has been “fixed” (its pattern output has become known, either because the pattern instance is a bottom-level pattern instance corresponding to an input/output that has already occurred, or its pattern inputs are all from pattern instances with known pattern outputs), it is only retained as long as there is at least one pattern instance using it as a pattern input which itself does not yet have a known pattern output. When this condition is no longer met, the pattern instance is erased.

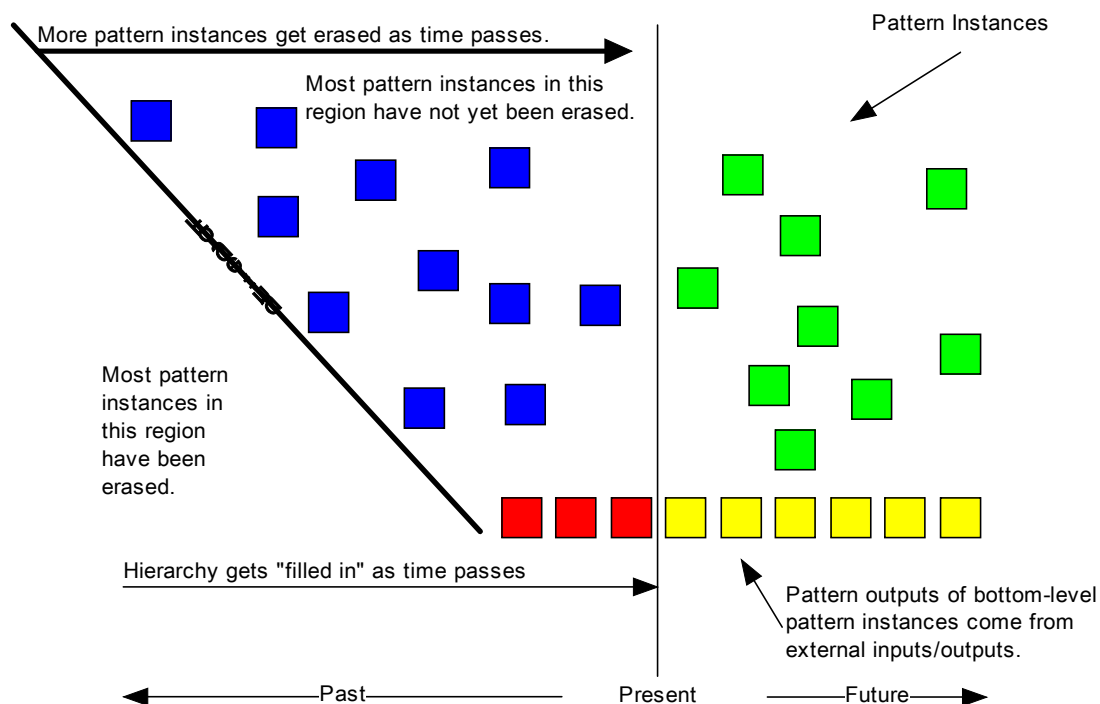
(That is to say, once a pattern instance’s pattern output value has become known with certainty, we only keep it as long as it is still relevant in determining the pattern outputs of other pattern instances with as yet unknown pattern output values. When there are none, we erase it.)

One thing to note about the basic forgetting procedure is that it does not involve any kind of weighing up of the benefits of retaining a pattern instance against its cost in computing resources. The process is clearer-cut: When a pattern instance is erased there is *zero* benefit in retaining it, because it can no longer affect *anything*.

4.4 The General Effects of Forgetting on the Hierarchy

With reasonably sensible patterns, we should expect low-level pattern instances to tend to be used by other low-level pattern instances which have pattern inputs corresponding to external inputs/outputs or other low-level pattern instances that occur or become fixed over a short period of time. For example, we would not expect an input corresponding to a single pixel at some instant from a video camera to be used as an input by a pattern instance which has another pattern output that will not become known until ten years later: We would expect any pattern instance using this as a pattern input to become fixed within seconds, or even less.

This should give us idea of how a hierarchy with appropriate patterns will tend to behave with such a forgetting procedure in use. In general, the higher the level of a pattern instance, the longer it will tend to persist in the hierarchy after being fixed. Pattern instances on the bottom level will tend to persist for the least time after being fixed, while pattern instances which are least directly dependent on the bottom level will tend to last longest. (See Figure 3, on page 16.)



Key

- Bottom-level pattern instances corresponding to external inputs/outputs that have already occurred. Each of these is "fixed": Its pattern output value was assigned, permanently, when the relevant input/output event occurred.
- Pattern instances which depend, directly or indirectly on bottom-level pattern instances for inputs/outputs which have already occurred. Each of these is also "fixed": Its pattern output value was assigned, permanently, when all the pattern instances being used for its inputs were assigned pattern output values.
- Bottom-level pattern instances corresponding to external inputs/outputs that have yet to occur. Each of these has not yet been "fixed": The pattern output value will be assigned when the relevant input/output event occurs.
- Pattern instances which depend, directly or indirectly on bottom-level pattern instances for inputs/outputs that have yet to occur. Each of these has not yet been "fixed": Its pattern output value will be assigned, permanently, when all the pattern instances being used for its inputs have been assigned pattern output values.

Figure 3: General Effects of Forgetting on the Hierarchy

In the above diagram, the vertical line labeled "Present" should be regarded as moving from left to right as time passes and inputs/outputs occur and bottom-level pattern instances become fixed, in turn causing higher-level pattern instances to become fixed. The sloped "Forgetting" line should also be thought of as moving from left to right. This line does not represent some process that is applied to the hierarchy: I am not saying

that pattern instances are automatically erased when they cross this line. Rather, this line is intended to give a general idea of the likely effects of the basic forgetting procedure – pattern instances being erased, and tending to be erased more quickly, after being fixed, the more directly they are connected to the bottom level of the hierarchy.

4.5 Basic Forgetting as a “Baseline” System

We may want the behavior of the system to differ from that in the basic forgetting procedure: We may want it to do more or less forgetting than such a basic method, but the basic method just described will serve as a kind of “baseline” for forgetting – the forgetting that will occur in the absence of any more complex requirements.

4.6 An Example of Basic Forgetting

I will now give an example of how the basic forgetting procedure works. The following discussion will relate to Figure 4, below.

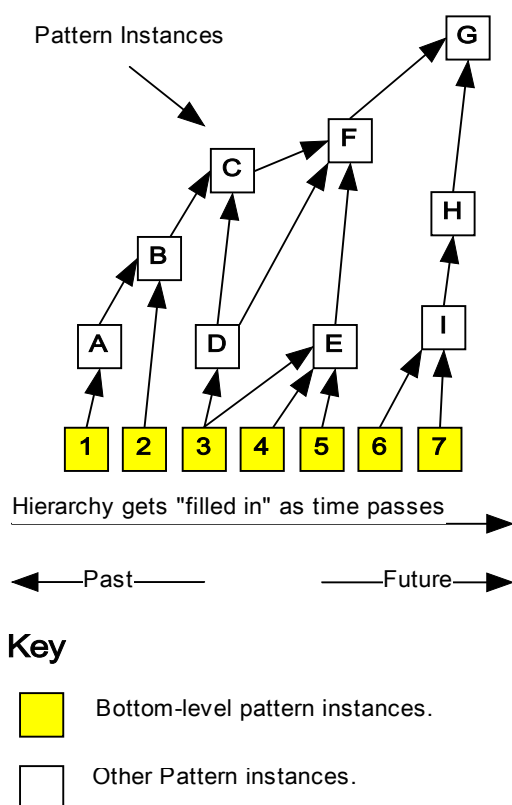


Figure 4: An Example of Basic Forgetting

Initially, none of the pattern instances are fixed.

An Attempt to Generalize AI – Part 3: Forgetting

When the input/output event corresponding to Pattern Instance 1 occurs, Pattern Instance 1 is fixed. Pattern Instance A uses only Pattern Instance 1 as an input, so Pattern Instance A is fixed. Pattern Instance 1 is not used as a pattern input by any other pattern instances, so Pattern Input 1 is erased. Pattern Instance A cannot be erased yet, even though it is fixed, because it used as a pattern input by Pattern Instance B, which is not yet fixed.

When the input/output event corresponding to Pattern Instance 2 occurs, 2 is fixed. B uses only A and 2 as pattern inputs, both of which are now fixed, so B is now fixed. B is the only pattern instance using A as a pattern input, so there is no more need for A and it is erased. 2 is not used as a pattern input by any pattern instance except B, which is now fixed, so 2 is erased.

When the input/output event corresponding to Pattern Instance 3 occurs, 3 is fixed. This causes D to be fixed. This in turn causes C to be fixed, as C needs inputs from Pattern B and D, which are now available. B is no longer needed, as it was only needed as a pattern input by C, which is fixed, so B is erased. 3 has to be retained for now, because it is still needed by E, which is not yet fixed. C and D have to be retained for now, because they are both needed by F, which is not yet fixed.

When the input/output event corresponding to 4 occurs, 4 is fixed. 4 still needs to be retained, however, as it is used as a pattern input by E, which is not yet fixed.

When the input/output event corresponding to 5 occurs, 5 is fixed. E is dependent on 3, 4 and 5, which are now fixed, so E is fixed. 3, 4 and 5 are no longer used by any unfixed pattern instances, so are now erased. F is dependent on C and E, which are now fixed, so F is fixed. C, D and E are only used by F, which is now fixed, so C and D are erased. F has to be retained, for now, as it is used by G, which is not yet fixed.

5 More Sophisticated Forgetting

5.1 Why We May Want to Use *Less* Forgetting

A simplifying assumption I made earlier was that no patterns are being added to the system, as they would be in any working system.³ That assumption was not absolutely necessary, but it avoided a complication potentially caused by adding patterns.

We are erasing pattern instances because they are not needed by any other current pattern instances with unknown pattern outputs, but this ignores any that we may add to the system later. We may erase some pattern instances and then add a new pattern to the hierarchy which, if it had existed before, would have required those pattern instances to be retained, by having pattern instances which would still be using them for inputs while not yet being fixed. This issue may not be a serious one, though, as it is self-correcting. As inputs/outputs continue to occur, and more pattern instances become fixed, these will replace the ones that were erased, and the new pattern's pattern instances can now hold onto these by requiring them as inputs for as long as necessary. Further, with a lot of pattern instances in the hierarchy, pattern instances will already be retained for as long as other, as yet unfixed, pattern instances need them. If a pattern's behavior is typical we might expect its pattern instances to have the same kinds of requirements as those of other patterns. For example, if a particular pattern instance is used by a number of other pattern instances and is going to be erased ten seconds after being fixed, we might consider it unusual if a new pattern is introduced into the hierarchy which sets up a pattern instance demanding that it is retained for ten weeks.

Nevertheless, this does mean that some patterns may be initially limited because the pattern instances already fixed by past inputs/outputs which its own pattern instances would have used are gone. This could cause a delay before more pattern instances can become fixed to replace them and the pattern starts to become useful. We may want to reduce such an effect by moderating the forgetting process.

5.2 Why We May Want to Use *More* Forgetting

When a pattern instance has been fixed, it remains in the hierarchy until there are no other pattern instances, themselves as yet unfixed, that use it as an input. This means that one pattern instance can “hang onto” another pattern instance, preventing it being erased. It may be that this causes pattern instances to be held in existence for too long. This may seem a strange statement, because the forgetting procedure I described causes pattern instances to be retained as long as they are needed, so how could we erase pattern instances sooner? There are possible ways around this problem, if we really need them.

³ Section 4.1: Simplifying Assumption on page 13.

5.3 Using *Less* Forgetting

If we want less forgetting, we could alter the existing process in a number of ways. One way is simply to scale it down by some proportion. Whenever a pattern instance is fixed, a certain amount of time (or a certain number of input/output events) is going to elapse before the pattern instance is no longer needed and the pattern instance is erased. We might multiply this time (or the number of input/output events) by some constant to get an extended time (or number of input/output events) for which we actually retain the pattern instance. We could work out this time when the pattern instance is about to be erased, and then extend the time by some amount, or we could look at the pattern instances using it as pattern inputs and work all this out beforehand: The distinction is trivial.

Example

Suppose we use a multiplier of 1.5 to extend the time for which a pattern instance is retained. A pattern instance was fixed 60 seconds ago, and it is now about to be erased. 1.5×60 seconds = 90 seconds, so the pattern instance is actually going to be retained for 90 seconds after being fixed, instead of the 60 seconds it would have got.

This should show what I meant, earlier, when I said that the basic forgetting procedure is a baseline: We are using it to give us a “default” erasure time here, and then making our adjustments on top of that.

This is only one approach that could be used.

5.4 Using *More* Forgetting

To retain pattern instances for less time than they would be retained by the basic forgetting procedure, we might use the approach described previously⁴, but making the multiplier a fraction of the time for which a pattern instance would last after being fixed.

Example

Suppose we use a multiplier of 0.5 to reduce the time for which a pattern instance is retained. A pattern has just been fixed and, if the basic forgetting procedure were applied, it would last for 30 seconds before erasure. 0.5×30 seconds = 15 seconds, so the pattern instance is actually going to be retained for 15 seconds after being fixed, instead of the 30 seconds it would have got.

⁴ Section 4.3: The Basic Forgetting Procedure, on page 15.

A problem with this is that it would not resolve the issue of what to do with the pattern instances, as yet unfixed, using a pattern instance that has been erased early as an input. We might think of doing this by erasing a pattern instance, but not before the inputs that it was providing to other pattern instances were stored with those pattern instances, but this would just be cosmetic. After a pattern instance is fixed, all that is left of it is its pattern output value anyway, and if this is retained in any form at all, even as a stored input value in another pattern instance, the pattern instance is, for all practical purposes, still in existence.

If we want more forgetting, we might do it by fixing some pattern instances early, if their final pattern output values are known with *almost* complete certainty, even if those pattern instances have pattern inputs which are not yet known. The advantage of doing this would be that such pattern instances, if they had some pattern inputs corresponding to pattern instances that had already been fixed, would no longer need those pattern instances, potentially allowing them to be erased. A disadvantage is that pattern instances would occasionally be fixed with incorrect values, but by increasing the amount of confidence that is required to do this, this problem can be reduced to an acceptable level.⁵

5.5 Still More Sophisticated Forgetting

Still more sophisticated approaches are possible, but they should all work by using the basic forgetting procedure described previously⁶, as a baseline, increasing or decreasing the extent to which this procedure is implemented as appropriate. Some approaches might work by adjusting the extent of forgetting that occurs based on how high-level a pattern instance is. Other approaches may make adjustments for individual patterns, or even individual pattern instances, based on various features of the hierarchy. Such approaches might take account of the hierarchy's degree of success in making accurate predictions, the uncertainty in the predictions its makes or the extent to which some adjustment in the forgetting procedure changes the attained, or predicted, evaluation function score (EFS) values.

⁵ Considerations like this might suggest the idea that the “fixed” pattern instances are just a special case of pattern instances in general, and we can just use a general approach, but I will use the idea of “fixing” pattern instances for now: I think it gives a view of the system which is easier to understand.

⁶ Section 4.3: The Basic Forgetting Procedure, on page 15.

6 Human Experience of Memory and Forgetting

Forgetting is something that we need to build into an AI system, but it is also something which we experience when we cannot recall events, people or things we have read and attempted to memorize.

The basic forgetting procedure described here has some similarities, and some differences, with our experience of forgetting. If you think about Figure 3, on page 16, it will tend to make the memory of the past have a gradually “lower resolution”, as the lowest-level pattern instances are taken out of the hierarchy first, and I think this is how things tend to seem to us. It should be noted, however, that this does not mean that the hierarchy itself somehow “tries”, and fails, to remember the past in a detailed way: Pattern instances are removed when they are an irrelevancy anyway – when they no longer play any part in what the hierarchy does.

Our experience of forgetting is one of sometimes being able to recall “forgotten” information, whereas the basic forgetting procedure erases it. We should be careful about what we infer from this, however. All of our conscious experience is due to pattern instances at relatively high levels in the hierarchy: “You” are, in fact, an object in a model. Any experience of forgetting that you have is occurring in the “you” that is being generated by the modeling in the hierarchy, and so it is not quite the same as “architectural-level” processes in the hierarchy itself. As an example, suppose you broke a vase seventeen years ago, and you can now, after all this time, remember the smallest detail of the pattern on that vase. One interpretation of this would be that very low-level pattern instances have been held onto for all this time. Another interpretation of it would be that your memory of the event, including your memory of the smallest details of the vase’s pattern, is an *experience* occurring at a much higher level than the original pattern instances on which it is based. When you “forget” something, this may not mean that the information has been really removed, irreversibly, from the hierarchy. It may be that a lot of information is removed from the hierarchy, but can still be inferred by running the kinds of modeling processes described on the information that remains.

For example, suppose you meet someone called “Fred Smith” at a party. Years later, you “remember” Fred Smith when he is mentioned in a conversation. It may be that your brain did not explicitly store the fact that the man’s name was “Fred Smith” in some simple way all this time. Instead, it could be that your brain’s hierarchy, when generating “you” as an object in its model, used the limited data that you have about the party to determine that the best model it could make is one in which your previous behavior is explained by modeling a person – “you” who remembers meeting Fred Smith at that party. This means that a lot of our “memory” of the past may be not too different from our probabilistic predictions of the future. We have limited information about both, and anything we work out is statistics.

This matches some views that human memory may be extremely pliable. For example, suppose someone is exposed to some experience. The low level description of that experience is ultimately lost, due to the basic forgetting procedure. Now, suppose we later contrive new experiences for that person in such a way that the hierarchy in that person's brain finds it easier to "explain" things by modeling a "self" that had experiences with a low level description that does not match reality. This person would now have a false memory of reality, but it would be absolutely persuasive to the subject, because it would be no different from other memories. They would all be the memories of a "self" produced by statistical modeling in a hierarchy.

As an analogy, imagine an author who reads about the life of some historical person. The author looks at what happened to this person and what he/she did. He/she then writes a "dramatized" account of the person's life – a book written in the style of a novel describing these events. The book describes what this character was thinking, what this character remembered from previous experiences, etc. The author is not entirely sure how the real-life person really thought about things, and what he/she remembered each day, but he/she has made his/her best guess, based on what the person did. Your brain is doing the same thing to generate you.⁷

All this means that conscious memory and what is going on at an "architectural" level in the hierarchy are somewhat different things. However, while our experience of forgetting is not necessarily the same "forgetting" as that in the basic forgetting procedure described here, some correspondence should be expected. Often, when you forget something, it will be because a "self" which has forgotten it is the simplest explanation of a history which includes corresponding pattern instances being removed from the hierarchy.

⁷ In the previous article, when discussing this idea that the "self" may not be as integral to the brain as widely believed, I mentioned previous work by Thomas Metzinger.
Metzinger, T. (2003). *Being No One: The Self-Model Theory of Subjectivity*. Cambridge, MA: MIT Press.
Metzinger, T. (2009). *The EGO Tunnel: The Science of the Mind and the Myth of the Self*. New York: Basic Books.

7 Conclusion

The two previous articles in this series, *An Attempt to Generalize AI - Part 1: The Modeling System* and *An Attempt to Generalize AI - Part 2: Planning and Actions*, described a hierarchy based on patterns, and how it could be made to learn and perform intelligent actions. The hierarchy, as described in the previous articles, is idealized. One issue with it is that pattern instances are retained permanently, long after they are relevant to what is happening in the hierarchy. Another issue is that a huge number of pattern instances are explicitly represented, and not all these will be relevant. Both of these issues would cause inefficient use of computing resources in any working system. This article has dealt with the first issue: pattern instances being retained permanently in the hierarchy.

When a pattern instance's pattern output is determined, directly or indirectly, by any future inputs/outputs, it is described probabilistically. When the pattern instance's pattern output depends, directly or indirectly, only on inputs/outputs that have already occurred, the pattern instance has a known pattern output value and is said to be "fixed". Importantly, once a pattern instance is fixed, and its pattern output value has been assigned, that value can never change: The fixed pattern instances are all frozen. Fixed pattern instances are needed to provide statistical information, used in *statistics application* to assign probabilities to as yet unfixed pattern instances, but this information can be obtained as soon as a pattern instance is fixed, and does not need obtaining from it again after that. Fixed pattern instances are also needed because they may be used as pattern inputs by other pattern instances; however this only matters if these other pattern instances are not fixed: If a pattern instance is used as an input by another pattern instance that is already fixed, then this is irrelevant.

This can be used to specify a basic forgetting procedure, as follows:

When a pattern instance has been "fixed" (its pattern output has become known, either because the pattern instance is a bottom-level pattern instance corresponding to an input/output that has already occurred, or its pattern inputs are all from pattern instances with known pattern outputs), it is only retained as long as there is at least one pattern instance using it as a pattern input which itself does not yet have a known pattern output. When this condition is no longer met, the pattern instance is erased.

(That is to say, once a pattern instance's pattern output value has become known with certainty, we only keep it as long as it is still relevant in determining the pattern outputs of other pattern instances with as yet unknown pattern output values. When there are none, we erase it.)

A procedure like this will tend to cause pattern instances to be erased sooner at lower levels of the hierarchy.

There may be situations in which less or more forgetting is desirable. Less forgetting may be wanted to ensure that there is an instant supply of pattern instance data for new patterns. More forgetting might be wanted if fixed pattern instances, or ones known with almost complete certainty, are consuming too much computing power. Less forgetting might be achieved by multiplying the time that a given pattern instance would last, after being fixed, by some constant, or increasing the time for which it lasts in some other way. More forgetting might be achieved by fixing pattern instances early, if the degree of uncertainty in their pattern output values is low enough, thereby removing the need to retain other pattern instances because they are using them as pattern inputs.

We might consider all this in the context of our own experiences of memory and forgetting. We should be careful about doing this, however. In the view being described here, the “self” is not a feature of the hierarchy’s architecture, but is just an emergent object in the hierarchy, like everything else, produced because it is the best explanation of the system’s previous behavior. This means that when you consciously “remember” something it is the person projected by the hierarchy as part of its model who is doing the remembering, and that is happening because a person doing the remembering is best explanation for the system’s previous input/output behavior. This can be considered analogous to an author constructing a narrative of a character’s internal thoughts, feelings and memories to fit a historical record of a person’s behavior. The same applies to your experience of forgetting: When you experience “forgetting”, it is the “self” being generated by the modeling system that is doing the forgetting.

While experiential forgetting is not necessarily the same as the basic forgetting procedure described here, some correspondence should be expected. Often, when you forget something, it will be because a “self” which has forgotten it is the simplest explanation of a history which includes corresponding pattern instances being removed from the hierarchy.

I have also commented on the construction specification for patterns. This has previously been described as directing the insertion of new pattern instances into the hierarchy, but it could be done the other way round, with pattern instances being set up. Maybe randomly, and then added to various patterns dependent on how well they match the construction specifications of those patterns. This could result in pattern instances having varying degrees of membership of patterns.

8 Bibliography

Almond, P. (2010). An Attempt to Generalize AI - Part 1: The Modeling System. *paul-almond.com*. Retrieved 27 February 2010 from <http://www.paul-almond.com/AI01.pdf>. (Also available at <http://www.paul-almond.com/AI01.doc>.)

Almond, P. (2010). An Attempt to Generalize AI - Part 2: Planning and Actions. *paul-almond.com*. Retrieved 6 March 2010 from <http://www.paul-almond.com/AI02.pdf>. (Also available at <http://www.paul-almond.com/AI02.doc>.)

Metzinger, T. (2003). *Being No One: The Self-Model Theory of Subjectivity*. Cambridge, MA: MIT Press.

Metzinger, T. (2009). *The EGO Tunnel: The Science of the Mind and the Myth of the Self*. New York: Basic Books.