

Planning As Modelling in AI: A Revised Description

By Paul Almond, 27 April 2007

Website: www.paul-almond.com
Email: info@paul-almond.com

© Copyright Paul Almond, 2007. All Rights Reserved.

Planning As Modelling in AI: A Revised Description

By Paul Almond, 27 April 2007.

Introduction

This article describes the *planning as modelling* approach to planning in artificial intelligence (AI). This uses an AI system's modelling system to produce probabilistic *predictions* of future behaviour that are equivalent to *planning* of future behaviour.

The article combines concepts from previous articles [1,2,3,4,5,6] (see Appendix 6) about planning as modelling.

A summary of planning as modelling is in Appendix 1.

How Planning As Modelling Works

Main Idea

Planning as modelling uses an AI system's modelling system to perform both modelling and planning without needing a separate planning system.

Planning as modelling also uses prioritization control outputs – special pseudo-outputs generated by the system in the same way as outputs, but which instead of acting on the external world, act on the modelling system as instructions to control prioritization of its use of computing resources.

Input and Output Events

Inputs and outputs of the AI system are considered in terms of *input events* and *output events*. An input or output event is the occurrence of an input or output at some instant with a specific value. Inputs and outputs in many systems would take values of “0” or “1” but could take other values.

The history of the AI system's experiences and behaviour is a sequence of input events and output events.

For past input events and output events, the input or output values are known. For future input events and output events the values can only be known probabilistically.

Modelling does not distinguish between inputs and outputs

The modelling system observes past input events *and output events* of the AI system and makes probabilistic predictions of the values for future input events *and output events*. *The AI system is predicting what will happen in reality and its own behaviour*: planning as modelling makes no

distinction between them as the system's observations of its own outputs are treated as just more inputs about which predictions can be made.

No Special “Self-Modelling” Feature

This lack of distinction between inputs and outputs within the modelling system is total. As far as the modelling system is concerned, past input and output events are merely observations used to make predictions and it does not need to know which are inputs and which are outputs when making predictions. The system has no clever, special feature for “representing itself” or “modelling its own behaviour” when predicting its own future outputs. It does not need to “know” that, along with modelling its future observations, it is modelling its own behaviour. In planning as modelling, self-modelling is an inevitable result of a system observing its own outputs.

The modelling system is informed of input and output events

When an input event or output event occurs, its input or output value becomes known and the modelling system is informed about its occurrence and its values. The modelling system is therefore continually informed about the inputs *and* outputs that occur for the AI system.

How the Modelling System Makes Predictions

The modelling system's predictions are limited to producing probabilities for future inputs and outputs. These predictions are based solely on the inputs and outputs that have already occurred; that is to say on the modelling system's observations of inputs, and its own outputs, that have occurred in the past. How the modelling system obtains predictions about future inputs and outputs from previous inputs and outputs is not part of planning as modelling.

Whether or not the modelling system even stores the information that it is given about input and output events as they occur, whether directly or (more likely) in some abstracted way is not an issue for planning as modelling. The modelling system is simply informed about the input and output events as they happen (or hypothetically) and will use this information to such an extent that its design requires it, storing it, abstracting it or discarding it, to make probabilistic predictions of future input and output events on demand.

The modelling system can be hypothetically informed of input and output events

The modelling system is not restricted to being informed about input and output events that have actually happened: it can also be informed hypothetically. The modelling system could be made to simulate hypothetical futures by informing it about input and output events that have not yet occurred, using possible future values for these events. This is done in planning as modelling, as is explained shortly.

Use of the modelling system for simulation necessitates the capability of restoring it to previous states: if we hypothetically inform the modelling system that a particular input or output event has occurred with a given value then we need to be able to “rewind” the modelling system to a previous state before it was informed of this.

This is equivalent to the way in which the state of a chess board is dealt with in chess algorithms. A tree search simulates possible future moves but the board is not permanently altered.

One way of doing this is to have an “undo” facility which reverses the effects of the most recent act of informing the modelling system about an input or output event which has not yet been undone.

The Output Vectoring System

When an output is needed a separate system, the *output vectoring system*, searches for the best output. The search is of the set of possible future sequences of inputs and outputs that the AI system can receive and make. This can be a tree search – as with chess algorithms [7]. In such a tree search each node would correspond to a particular input or output event occurring at some instant of time and each branch to a particular value for an input or output event and the search tree would explore possible future inputs and outputs in chronological order.

Traversal of the Output Vectoring System’s Search Tree

The search does not equally prioritize all possible sequences of future inputs and outputs. Instead, *the search is constrained by the modelling system’s predictions*. More likely futures are explored in greater detail than less likely ones. The likelihood of a particular future is obtained by multiplying together all the probabilities of the individual inputs and outputs occurring as they do in that future and the modelling system provides these probability values.

With a tree search the constraint from the model can be applied using a *running probability*. The tree search computes a *running probability* at each node. At the start of the search tree this has a value of 1. The running probability represents the likelihood of a particular future - or the probability that a sequence of input and output events will occur to produce the situation described corresponding to a particular node of the search tree.

Each time the search tree goes down a new branch for an input or output event occurring with a particular value then the modelling system is asked to provide a probability value for the event occurring with this value and the running probability is multiplied by this probability to get the running probability for the new node. The modelling system is then informed (hypothetically) that the corresponding input or output event has occurred with the corresponding value.

Paths through the search tree are prioritized according to the running probability: a path through the search tree is never examined if one with a lower running probability is discarded. New branches of the search tree are only extended from a node provided that the running probability for that node does not exceed the *cut-off probability*.

When the cut-off probability prohibits extension of search tree branches, terminal nodes of the search tree are evaluated by a situational evaluation function like those in chess programs [8] which gives a score indicating the desirability of the situation described by the model. This score is “backed up” the search tree to previous nodes.

The Situational Evaluation Function

The situational evaluation function returns a score for the situation described by a given state of the model. It does not need to deal with any abstraction in the model: this is not practical as it would require the evaluation function to be as sophisticated as the model. More likely, the situational evaluation function would examine previous inputs (and possibly outputs) – most likely very recent ones – to determine what sort of situation the AI system is in. The modelling system, being informed of input and output events as they occur can provide this data. The modelling system may just provide the history of input and event values or it may maintain separate data structures, containing information derived from previous input and output events, to provide information to the situational evaluation function, as in the chess example in Appendix 4. This information could also be stored outside the modelling system, but it would be need updating with the model. Whether we regard it as part of the modelling system or not is really semantics.

Dealing with Backed-Up Scores

At terminal nodes, scores are produced by the evaluation function and backed-up to higher nodes, meaning that the score is made available to the processing at the node immediately above it.

When scores are backed-up to a node, processing occurs on them to determine a single score to be assigned to that node as if it were a terminal node. This score is in turn backed-up to higher nodes. A similar approach is used in chess programs [7].

Backed-Up Scores to an Output Event Node

The score to be assigned to an output event node is the highest score backed-up to it from lower nodes. The output value associated with the highest score is also backed-up because the selection of a score and output value at an output event node is equivalent to modelling the system's own behaviour in the situation corresponding to that node, and the system is to act to maximize the score at a node. We will not actually need to know what the output value was except for the first node, when we need to determine what output should be made, so alternatively the actual output value could just be returned from the first node and only the scores backed-up from lower nodes.

Backed-Up Scores to an Input Event Node

The scores backed-up to an input event node correspond to the utility of various outcomes, with probabilities, and the system cannot control which outcome (input value) occurs, yet it must assign a score to that node. The most obvious approach is to assign the mean (average) of the scores backed-up to an input event node to that node. Other approaches could be more optimistic – backing up the highest score – or pessimistic – backing up the lowest score, or some compromise between using the mean score and an optimistic or pessimistic approach, depending on the kind of behaviour wanted.

Returning an Output Value from the Output Vectoring System

The purpose of the output vectoring system is to return the best output to make for the current event at the time that its search starts. The top node in the search tree corresponds to the current event so when we select the highest score backed-up to this node as the score to be assigned to it we need to select the particular output value associated with this score and return it from the output vectoring system as the best output.

Prioritization Control Outputs

Why Prioritization Control is Needed

However the modelling system is constructed, computing resources need prioritizing. The modelling system will need to decide what input data is most relevant in generating the probabilistic predictions of inputs and outputs and which of many possible computations are most relevant. Some computations will be based on intermediate results and the decision about which intermediate results to use could be complex. All of these decisions may vary from time to time and the modelling system needs to adapt the way it computes accordingly.

In summary:

The modelling system needs to think only about relevant things.

This issue of keeping a computing system “focused” like this is often known as the *carpet texture problem*.

Some way is needed of controlling prioritization of computing resources in the modelling system – what it concentrates on – at any time.

The term “prioritization” is being used in a different way here than earlier in the article. It refers here to the relative importance of different types of processing in the modelling system rather than the relative importance of different possible sequences of future inputs and outputs being examined by the output vectoring system as previously.

How Prioritization Control Outputs Work

Some of the AI system’s outputs are designated as special *prioritization control outputs*. Prioritization control outputs do not control events in the outside world. Instead, they control prioritization of computational resources within the modelling system.

Apart from what they control, prioritization control outputs are dealt with in the same way as other outputs:

- There is no special planning involving them – the existing planning system involving interaction of the modelling system and output vectoring system encompasses the prioritization control outputs.

- The prioritization control outputs are observed by the modelling system along with the other conventional outputs, so that the modelling system can use them in making its predictive model.

This means that the AI system is making outputs to control its own modelling system just as it would manipulate its outside environment. In a way, the modelling system *becomes* part of the outside environment, an idea which I called *AI as a boundary system* in a previous article [5].

The details of how a given prioritization control output will affect prioritization in the modelling system will need to be specified. These are dependent on specific details of the modelling system and beyond the scope of this article.

A more detailed description of the services that the modelling system could provide to allow planning as modelling to work is in Appendix 2. Example Pascal computer programs showing how planning as modelling could work are in Appendix 3. An example of how planning as modelling could be used in a chess program is discussed in Appendix 4.

Why Planning as Modelling Works

Future behaviour is based on past behaviour

The constraint from the modelling system causes the output vectoring system to ignore futures reached via unlikely sequences of input and output events. Lack of distinction between inputs and outputs in the modelling system means that this modelling is based on the system's experiences *and behaviour*. If the modelling system considers some future behaviour unlikely then it will result in an unlikely future that is cut off early in the search tree. Assuming that the modelling system has performed competently in the past, then a model of the system's behaviour, based on this past behaviour, should predict competent behaviour. It may seem that we could directly use the most likely behaviour predicted by the modelling system as the basis of future behaviour, but this would fail because we need to establish a pattern of competent behaviour to start with. The output vectoring system does this by doing some searching, constrained by the modelling system's predictions, in an attempt to optimize this behaviour. It also has a role in providing stability.

One way of thinking about this is that the modelling system suggests a "path" through the future, based on the path that the system has previously followed. The output vectoring system then "sniffs out" other paths on either side of this path looking for slightly better paths. When these are found they are used and become part of the past behaviour which is used to model future behaviour.

Any performance increase "earned" by the output vectoring system becomes the default behaviour

Suppose that the output vectoring system finds some behaviour that, though not the most likely behaviour as predicted by the modelling system, seems better. Some computational work by the output vectoring system was needed to "earn" the knowledge about this behaviour. The system will select this behaviour. Once performed, this behaviour is part of the system's history of previous behaviour and contributes to the pattern of past inputs and outputs that the system observes. After

this kind of behaviour has occurred a number of times, the modelling system will start to *predict* it. It will then become the most likely behaviour and will correspond to the preferred route through the search tree (or at least a likelier route than it was before). Behaviour that once took considerable searching to find is now the *default* behaviour (or close to it) and is available with little computational expense in the output vectoring system. The computing resources in the output vectoring system are now freed from finding this behaviour to be spent on exploring nearby paths to try to improve on it.

This requires considerable computing in the modelling system, but that is required anyway. Planning as modelling merely makes planning primarily a modelling system process.

Only important possible futures are investigated

In planning as modelling, futures reached by a sequence of likely input/output events are considered more important, therefore meriting more investigation. A more likely future is more important because:

- With regard to outputs, it is more likely to be caused by the system. A possible future being considered likely means that modelling of the system's own behaviour makes it likely that *future behaviour* will cause it – making it likely to be more desirable.
- With regard to inputs, it will have a greater effect on any averaging of the future desirability of outcomes.

The modelling system is doing the planning

This may seem to be just the uninteresting addition of a planning system to a modelling system, but there are two important features:

- The modelling system makes probabilistic predictions of the system's inputs *and outputs*.
- These predictions constrain the search involved in “planning”.

The output vectoring system itself is doing little of any complexity. Almost all of the sophistication in this process comes from the modelling system.

The modelling system is doing the planning.

The modelling system's *predictions* of the AI system's future behaviour are *directing* its future behaviour.

Planning is prediction.

The argument for this will become stronger later when we consider *output vectoring system marginalization*.

The Planning/Improvement Cycle

The way in which the system improves its behaviour can be thought of as the *planning/improvement cycle*. This cycle is not something that needs doing explicitly, but just a simplified description of how the system's outputs feed back into the model to control later outputs and the role of the output vectoring system in this.

The system does not really get into the planning/improvement cycle immediately, so the description below will include the leading up to the start of it.

1. The system initially lacks previous input or output events. All the probabilistic prediction values produced by the system give no information about its future, e.g. for a system with binary inputs and outputs they would all be 0.5. The output vectoring system achieves nothing and the system's behaviour is arbitrary.
2. Input and output events have now occurred and the modelling system contains patterns relating inputs to outputs. Any projection of future behaviour is based on arbitrary previous behaviour and is still useless. When the output vectoring system simulates inputs and outputs the modelling system can at least now give probabilistic predictions accordingly, because enough previous data exists to do this. This means that the output vectoring system should at least be able to assess the consequences of actions in terms of desirability. This is better than nothing and a slight improvement in behaviour should occur. The "planning" (in the output vectoring system itself), however, will be very short range, due to the lack of any effective constraint that is not arbitrary.
3. The system now has a model containing better behaviour from Step 2. The modelling system makes a projection based on this previous behaviour. This projection, expressed as future input and output event probability values, constrains the output vectoring system's search. The modelling system itself is now having some influence on behaviour. There is no tendency in the modelling system to improve the behaviour (yet) but the output vectoring system will attempt to find the optimum behaviour, given the probability values from the modelling system. It manages to slightly improve on the previous behaviour in Step 2, because it has the same computational resources but now has the benefit of this constraint.
4. The system has previous behaviour that has been improving, so the model of its future behaviour will describe an improving system. The probability values from the modelling system will therefore constrain the output vectoring system in a way describing improved behaviour – even before the output vectoring system starts working. By performing a tree search the output vectoring system is able to improve on this behaviour slightly, so the improvement achieved now is better than the previous improvement.
5. The system's behaviour has not only improved in the past. Its *rate* of improvement has increased. The model therefore describes a system functioning in this way and the constraint in the predictions already describes a system which is improved more than last time. This constraint allows the system to improve more than last time, even without the output vectoring system doing any work. The output vectoring system does perform a tree search, however, and makes a very slight improvement on this. The rate of improvement has now been increased, and this will be reflected in future modelling of the system's behaviour by the modelling system without the output vectoring system having to provide it.

The planning/improvement cycle can be thought of as Step 5 repeating. There is a runaway aspect to this. Where I left it, the modelling system had not only acquired the tendency not only to improve the system's behaviour, but also to increase the rate of improvement of the system's behaviour.

This view is idealistic: limits on performance are discussed later.

Why Prioritization Control Outputs Work

Prioritization control outputs resolve the issue of how the system learns how to adjust the internal prioritization in its modelling system. It learns how to do this in the same way that it learns how to plan any other aspect of its behaviour.

Prioritization control outputs that inappropriately set priorities in the modelling system will cause it to spend its limited computing resources wrongly and work in a sub-optimally, with too much uncertainty in areas where more certainty was needed. This could happen for various reasons:

- All of the probabilistic predictions of inputs and outputs have a lot of uncertainty because processing has been wasted on computing intermediate results that have little effect on these predictions.
- Processing has been wasted on achieving an unnecessarily high standard of prediction for a small number of input and output events.
- Processing is being wasted on achieving a high standard of prediction for irrelevant input and output events.

If the modelling system behaves like this – and initially it will – it will be inefficient. The output vectoring system will encounter a lot of uncertainty in simulations of actions and predictions made by the system will not achieve high situational evaluation function scores. If, however, the system produces behaviour with better prioritization then the model predictions will have less uncertainty for those inputs and outputs where it matters and the system will be able to achieve high evaluation function scores.

This will affect the search process of the output vectoring system. When it tries prioritization control outputs that result in simulations by the modelling system with too much uncertainty to get good situational evaluation function scores, and also tries prioritization control outputs that result in simulations with less uncertainty that do allow it to achieve good scores, the prioritization control outputs that reduce uncertainty in useful ways will be found to be better and will be selected by the output vectoring system. The output vectoring system does not need to know that it is doing this by using the prioritization control outputs: in fact, it does not need to know which of its outputs are prioritization control ones and which are conventional. In this way, the output vectoring system slightly “nudges” the prioritization control outputs made by the system in the direction of better modelling. Once made, these prioritization control outputs are part of the system's behavioural history and will naturally play a part in the predictions of future behaviour made by the modelling system. In this way, the system will learn to organize itself. There is nothing special about this process: it is how the AI system learns any other type of behaviour.

Planning need not be simple

Using previous behaviour as a guide does not mean the modelling system is merely expected to generate future behaviour by simplistically copying previous behaviour. Basing future behaviour on previous behaviour means basing it on a *model* generated from past behaviour. The relationship between past and future outputs is merely that they are part of the same model: this model can have any degree of sophistication supported by available processing power.

Far from demanding that the system simply copy old behaviour, this actually allows it improve its behaviour. If the past behaviour shows a history of the system's performance in some task improving, and if there is enough of a history of inputs and outputs, then the most obvious model is one predicting further improvement for the system.

General Comments on the System

Planning Is Prediction

If the modelling system generates predictions, and actually does most of the system's planning, then what I am saying here is that planning is really prediction. The modelling system is doing the planning.

Though maybe seem strange, this is natural. In everyday life we have a good idea, from modelling, what other people will do next. If modelling can tell us what someone will do next, it follows that the person being modelled could also know what he/she is going to do next, given access to the same kind of modelling – making a good case that this is how people determine what to do next.

We are used to thinking of “making your mind up” as determining the optimum actions to make, given some model. In the context of this article it means something different. When you have not “made your mind up” about something you lack sufficient information to predict your actions with regard to it. When you experience the “making up of your mind” it means that you now have enough information in your model to predict what you are going to do.

Is it *really* the modelling system doing the planning?

There could be some debate about whether the modelling system is really directing the system's behaviour. The final behaviour is that which is produced by the output vectoring system's tree search and it could be argued that planning still occurs in the output vectoring system and the modelling system helps it by cutting off needless searching.

The modelling system should be viewed as doing the planning. Without the modelling system's constraint the planning process would never acquire any sophistication. It is improvements in the modelling system, due to a history of behaviour being acquired, that allow the output vectoring system to work. All the learning and sophistication is happening in the modelling system.

The modelling system does not just allow the output vectoring system to work more efficiently: it actually affects its results. By making the search concentrate on more likely futures, behaviour

leading to such futures, all else being equal, is more likely to score higher. For example, if we have two output values with probabilities of 99% and 1% of occurring then the more likely output value will have searches extending after it with a higher running probability and these will go deeper, because it will take longer to reduce the running probability, allowing them to spread out more and give more terminal nodes. The greater number of terminal nodes for the more likely output gives more opportunity for finding higher scoring terminal nodes, making it more likely that this behaviour will actually be selected.

Purpose of the Output Vectoring System

If planning occurs in the modelling system then the output vectoring system, despite superficially resembling a “planning system”, is not really the AI system’s “planning system”. This raises the issue of what the output vectoring system’s purpose is. It has two purposes:

- to slightly improve the behaviour “planned” by the modelling system’s predictions
- to provide stability for the system at a later stage

The predictions for the AI system’s future inputs and outputs are based on the AI system’s previous inputs and outputs. If the previous behaviour is desirable then future behaviour should also be desirable, but there is no reason why the behaviour should initially be desirable. The output vectoring system deals with this by trying to find optimum behaviour, within the constraint of the modelling system’s predictions. The output vectoring system’s search can improve on previous behaviour because it not only has the previous behaviour as a guide, but is performing a search as well. If it manages to find even slightly better behaviour then the system’s own observations of making the outputs associated with this improved behaviour will form part of the history of inputs and outputs used to constrain future searches by the output vectoring system.

An important aspect of this process is that the improvement itself will form a pattern of previous behaviour on which future behaviour will be modelled.

One way of thinking about this, and it is a little strange, is to regard the system as generating its future behaviour by modelling itself and the output vectoring system as finding ways of improving this behaviour so that when the system is modelling its future behaviour from previous behaviour it gets tricked into thinking it was smarter than it really was when it produced the previous behaviour.

Limits on Performance

The earlier description of the “planning/improvement cycle” may seem to be an attempt to get a “free lunch”. There is no “free lunch” in reality. Even with an inbuilt tendency to improve its own behaviour, the system is still dependent on how much abstraction the modelling system can provide. There will still be limits on this, one limit being the available computing power and another being *output vectoring system marginalization*, which can be seen to occur from the following consideration:

The description of the planning/improvement cycle was idealistic. In reality, this best describes the system’s early development. To see why this is the case, let us imagine that the modelling system

becomes very advanced. If it is really advanced, how can the output vectoring system possibly improve on its predictions and find better behaviour by using comparatively simple processing? If the modelling system is so good should it not have predicted this and included it in its predictions anyway? If this does happen, however, what happens when, even after this, the output vectoring system still takes these predictions as its constraint and tries to find improvements? Ultimately, what must happen is output vectoring system marginalization, a situation where the modelling system has anticipated so much that the highest probability outputs found by the modelling system are almost always those found to be best by the output vectoring system, so that its search simply fails to find anything. At this stage the output vectoring system is not contributing anything to the system's improvement and this strengthens the case that it is really the modelling system that should interest us. This does not mean that improvement of the system stops here, however: the pattern of previous improvement in the system's behaviour, extended into the future by the modelling system, should provide further improvement, subject to limits on available computing power. All it means is that the modelling system has "outpaced" the output vectoring system and has taken over.

At this stage the output vectoring system still has a role. Planning as modelling is statistical and small errors will contribute to the system's history of behaviour, which is the basis for later behaviour. The system would therefore randomly drift. The output vectoring system will stop this because if the system drifts enough to compromise the modelling system enough to make the output vectoring system no longer marginalized, the output vectoring system will once more start to improve on the modelling system's predictions, increasing the system's performance until it is marginalized again. The output vectoring system therefore has a long-term role in providing stability.

The Link Between the Model and Desirability

How is the system supposed to "know" to improve itself if its behaviour is based on modelling from its previous behaviour? This "direction" to the system's behaviour is given by the output vectoring system, which does test possible outputs according to desirability. The output vectoring system can be considered a link between the modelling system (which is doing the real planning) and the way that desirability of inputs is defined.

Prioritization Control and Integrity

We could fallaciously ask how, if the prioritization control outputs are all wrong, as they must be when the AI system is started up, we can know that use of the modelling system in simulations will cause low scores. If the modelling system is not working properly, what would stop it wrongly making predictions giving high scores and suggesting that the incorrect prioritization control outputs are good? This fallacy would be based on the idea that the prioritization control outputs control everything in the modelling system. This is not the case. Prioritization control outputs do not affect the integrity of the modelling system at all. The integrity of the modelling system must always be assured irrespective of what the prioritization control outputs are. The prioritization control outputs do control how the modelling system spends its computing resources and which aspects of the modelling are done in detail and which are represented by abstraction.

The Importance of Prioritization Control

Although the modelling system needs to ensure that the model has integrity without prioritization control, this does not mean that prioritization control has a minor role of “fine-tuning” a modelling system. Setting up the prioritization in a modelling system is a critical part of making the model itself.

For example, the simplest modelling system with integrity in a system with binary inputs and outputs could just spew out the value “0.5” for every prediction of a future input or output event. Such a system may have integrity, but there is nothing there. A more sophisticated modelling system lacking any kind of prioritization control may attempt to analyse everything. It would not get very far though: time constraints imposed by the need to act in the real world would limit such computation and the computation that did get done would be almost arbitrarily. By trying to analyse everything, such a modelling system would analyse almost nothing.

Without prioritization control, a modelling system lacks sophistication. Prioritization control is a deep part of what the modelling system is. An intelligent system is not just a modelling system, with some prioritization control helping it to think more efficiently. Rather, the modelling system lacks significant sophistication and is like a featureless block of stone out of which prioritization control carves a mind.

Obtaining a “Better View”

The following analogy gives an idea of how prioritization control outputs work, and shows why they need no special type of learning:

Imagine a robot which uses planning as modelling. It is capable of planning actions in the world – of manipulating the world to improve its situational evaluation function scores. Suppose there are easily-moveable obstacles blocking the robot’s view, and what is behind them may be relevant to the robot’s situation. We should not be surprised if the robot moves the obstacles. Doing so could reduce uncertainty in some aspect of its future predictions, allowing it to chart a path through these predictions that improves its score.

Suppose that after moving the obstacles the robot sees a computer scientist who offers to make some improvement to its modelling system. The robot should not need any special type of behaviour to evaluate this offer. If the scientist’s claim is correct, accepting the offer would involve making outputs that result in the system having better probabilistic predictions – just like the decision to move the obstacles.

In both moving the obstacles and accepting the scientist’s offer the AI system is simply making outputs that give it a “better view”. Whether this “better view” is achieved through making outputs that just alter the environment or making outputs that alter the modelling system in ways that allow better scores to be achieved is irrelevant.

Instead of the scientist we can now imagine the robot finding a tool kit and making the alterations to its own modelling system by itself and we can take this further. Ultimately, we are left with certain outputs that directly affect prioritization within the model system.

This gives a simple view of prioritization control outputs: as do-it-yourself brain surgery.

While specific details of how prioritization control outputs work within the modelling system need to be decided for any specific modelling system used, prioritization control outputs are the general way in which the carpet texture problem should be solved.

Prioritization Control, Irreversibility and Forgetting

There is no reason, in principle, why prioritization outputs should not be able to make irreversible changes to the modelling system, provided that these do not compromise its integrity; for example, prioritization outputs could order some detailed information in the modelling system to be replaced by abstraction. It should be noted that “irreversible” does not mean that changes made to the model at lower nodes in the search tree are still present when processing returns to higher nodes: it is a more specific meaning of “irreversibility” which applies when considering input and outputs following after each other in chronological order.

This is relevant with regard to the issue of storage of the historical data about past input and output events in the modelling system. Explicit storage of all this historical data – that is to say, storing the value of every input and output event – could require a lot of storage capacity and a greater problem may be that storage of this data will mean that it gets processed, potentially using a lot of the system’s resources. It is unlikely that the human brain explicitly stores all the historical data in this way. Such abstraction can be directed by prioritization control outputs which replace detailed historical data by abstracted versions of it when the benefits of abstraction in terms of reduced storage capacity requirements and processing outweigh any loss of accuracy in prediction. This would be *forgetting*. It would be valid for prioritization control outputs to do this provided that the integrity of the modelling system remained intact.

If the modelling system is caused to “forget” parts of the historical record of input and output events like this, the prioritization control outputs causing it are planned within the modelling system itself: forgetting is not being imposed from outside, but rather the modelling system is itself determining what it needs to forget as part of planning as modelling.

Self-Modelling as an Emergent Property

Planning as modelling lacks any special “self-modelling” feature. Instead, the system models reality in general, predicting its future inputs and outputs. This is not a claim that self-modelling is not required for AI. Rather, it is a claim that self-modelling is just an emergent property of the system, no different from the system’s modelling of anything else it observes in reality. The modelling system does not have any special “football game modelling” feature, but we would expect it to be able to model a football game: we can similarly expect it to model itself without a special self-modelling feature.

The Horizon Problem

How the Horizon Problem Occurs

If the system predicts that it is unlikely to behave so as to cause a particular future then that is equivalent to saying that, based on past analysis of the system's behaviour, the system will find that future undesirable. We expect less likely futures to have lower scores and when we find that this is so at a particular node we need not waste processing resources on looking further.

A problem possibly occurs, however, if the shallow search for an unlikely future actually returns a higher score than the deep search for a likely future.

For example:

Let us say that for a particular output being made in some hypothetical future situation, the modelling system gives a 99% probability of an output of “1” and a 1% probability of an output of “0”.

The prediction of a 99% chance of an output of “1” being made is equivalent to a prediction of a 99% chance that an output of “1” will be found the more desirable output, based on the system’s previous behaviour. There is, however, a 1% chance that an output of “0” will be found preferable.

Because of the different probabilities, the running probability maintained by the output vectoring system has different values as the output vectoring system’s tree search goes down each of these paths. The running probability will change further in various ways as branching occurs, but – all else being equal – the search down the “1” path will be deeper because it has “used up” less of the running probability.

What if the score from the shallow search down the “0” output path, backed-up to this node in the output vectoring system’s search tree, is *better* than the score backed-up from the search down the “1” output path? This would suggest that, although there was a 99% chance against it, the output vectoring system has determined that the better output to make at this node is “0”. The score for the “0” output was returned by a shallower search and could therefore be less reliable than the score from the search down the “1” path.

This could compromise the behaviour of the system. It could select a behaviour that its modelling system considers less likely, and therefore less likely to be desirable, because the short-term results of that behaviour seem better than the long-term results for the more likely behaviour, causing the behaviour to be favoured by the shallower search process for the less likely behaviour. The system is therefore experiencing a horizon problem.

Further information about the horizon problem is in Appendix 5 and a more detailed example of how the horizon problem can occur is in Appendix 5.1.

Horizon problems are well known in tree searches, for example in chess algorithms [9]. An obvious, and untenable, solution would be to make *all* the search deep. This would involve

discarding planning as modelling completely because the whole idea of it is to use the modelling system for planning by having it constrain a search for optimum behaviour.

Some reasons for which the horizon problem needs resolving are as follows:

- It causes the system's behaviour to be occasionally sub-optimal. Even if the system works reasonably efficiently, its efficiency should be maximized.
- The system's ability to work properly in many situations may give humans high expectations of it, yet the horizon problem could cause errors in situations where the correct behaviour would be natural to humans who would expect the system to work flawlessly. This could cause overconfidence in the system. It could also make it fit badly into a world designed for humans and human decision-making.
- The weakness caused by the horizon problem could be exploited by an intelligent opponent in some game or confrontational situation such as chess.

Resolving the Horizon Problem at Output Event Nodes

The Method

Dealing with the horizon problem with outputs involves detecting paths through the search tree where computation for certain output events involves unreliable scoring due to the horizon problem and repeating parts of the computation as if the probabilities for some output values are higher.

For a tree search, the probabilities for different output values indicate how deep the search will tend to be extended from the current output event node. For example:

Suppose at an output event node the running probability is currently 0.0012 and there are two outputs: "0" with a probability of 0.01 and "1" with a probability of 0.99. When the search tree goes down the "0" branch, the running probability is multiplied by 0.01, giving $0.0012 \times 0.01 = 0.000012$, and when it goes down the "1" branch it is multiplied by 0.99, giving $0.0012 \times 0.99 = 0.001188$. After going down these branches the search spreads out for different values of future inputs and outputs. Some of these paths will terminate quickly while others go deeper, depending on the probability values and their effect on the running probability and how quickly it reaches the cut-off value, but all else being equal, searches will tend to be shallower for the "0" value because it starts from this node with a much lower running probability.

The probability for an individual output value at a node therefore indicates the reliability of the backed-up score from the extension of the search for that output value.

We do not have to use the probability from the modelling system for determining the contribution of a particular output value to the constraint on a path through the search tree (that is to say, how close it is to being cut off). On detecting that a particular output value is being assumed to be the output that the system would make for a particular output event because the score backed-up from the search associated with it is higher, we can use a different, *higher* probability, for the purpose of determining the contribution of that output value to the constraint. This higher probability is the *effective probability*.

When using a cut-off probability with a search tree and the score selected at an output node is considered unreliable, due to being associated with a low probability output, we can repeat the extension of the search from the current node for that output value, multiplying the running probability instead by a higher probability value – a higher effective probability – deepening the search, to determine if that output value is still the best.

The Need for Availability of Secondary Searching At All Nodes

Such a secondary search needs to be available at each node of the output vectoring system's search tree – not just at the first node when selecting the very next output to be made.

At each search tree node the results from searches down different paths from that node are combined to give a single score, implying an assumption that the scores are equally reliable.

Considering output nodes, for example:

For a future output event node, backed-up scores for each possible value of that output are sent up to that node. The best scoring output value is selected as the output which would actually be made at that node, meaning that the score corresponding to that output value is regarded as the score which should be backed-up from that node.

This means that at every output event node the system effectively models its future choice of the best output to make in that situation by comparing the backed-up scores from the paths through the search tree following on from that node. If, however, the reliability of different searches varies depending on their depths then the scores backed-up to an output node for some output values could be less reliable than the scores backed-up for other output values and the selection of some output values based on them having the highest scores could be less justified.

Details of the Method

Step 1:

Let $\text{Probability}_{\text{Max}}$ be the probability value given by the modelling system for the most likely output value.

$\text{Probability}_{\text{Max}}$ does not change in subsequent steps.

Step 2:

Extend the search from this node for each possible output value as described previously for planning as modelling. Make a table showing, for each possible output value, the backed-up score and "Deep Search" – a flag (yes/no) value for each possible output value indicating whether the score for it was obtained using a deep search. A deep search is defined as one in which the search was extended from the current node, for that output value, with an effective probability of $\text{Probability}_{\text{Max}}$. The effective probability is the probability value assumed for that output value, for

the purpose of controlling the depth of the search, when extending the search from this node for that output value. (If a cut-off probability is used then, as the search goes down the branch for each possible output value, the running probability becomes running probability \times effective probability.)

Step 3:

If the highest score in the table corresponds to an output value for which the “Deep Search” flag is set (that is to say, it was extended from the current node with an effective probability of $\text{Probability}_{\text{Max}}$) then assume that the resulting score at this node is this score and that the output selected at this node is the output value corresponding to it.

If the highest score in the table corresponds to an output value for which the “Deep Search” flag is *not* set (that is to say, the search was extended from the current node with an effective probability *less* than $\text{Probability}_{\text{Max}}$) then extend the search from this node again for this output value, but with an effective probability of $\text{Probability}_{\text{Max}}$, setting the “Deep Search” flag for this output value to indicate that the search has been extended from the current node with an effective probability of $\text{Probability}_{\text{Max}}$, and update the score backed-up for this output in the table. (If a cut-off probability is used then, as the search goes down the branch for this output value again, the running probability becomes running probability \times the new effective probability.) Repeat Step 3.

Examples of use of this method for solving the horizon problem are in Appendix 5.2.

Alternative Implementation

The above method involves deepening the search by using the most likely output value’s probability as an effective probability, but a different effective probability value could be used.

Instead of using a flag (“Deep Search”) to indicate whether or not a search had been done with the higher effective probability, the actual value of the highest effective probability used for a given output value could be stored with it in the table. I avoided that here, mainly because of technical issues with rounding of numbers and comparison.

Special Case for Two Output Values

Planning as modelling will often be used in systems with only two possibilities for inputs and outputs – binary systems in which we can designate possible inputs or outputs as “0” or “1”. In a binary system, the above process can be used but the following, simpler process will do the same:

Step 1:

Extend the search from this node for output values of “0” and “1” as described previously for planning as modelling, with the effective probability for each output value being the probability from the modelling system. (If using a cut-off probability, multiply the running probability by the probability from the modelling system.) Store the score backed-up to this node for each such path.

Step 2:

If the higher score is that associated with the output value assigned the *higher* probability by the modelling system then assume that this output value would be selected in the situation corresponding to this node and assign this score to this node.

If the higher score is that associated with the output value assigned the *lower* probability by the modelling system then perform a secondary extension of the search from this node, down the branch corresponding to this lower probability output value only. Instead of using the probability given by the modelling system for this lower probability output value to control the depth of this secondary search, use the probability value for the *higher* probability output value as the effective probability. (If using a cut-off probability, multiply the running probability by the probability for the higher probability output value, so the search has the same depth.) Use the new backed-up score to replace the score previously calculated for the lower probability output in the shallower search. Compare the scores for the higher probability and lower probability outputs again. Assume that the higher scoring output value is the one that would be selected in the situation corresponding to this node and assign the corresponding score to this node.

Examples of use of this method to solve the horizon problem for binary systems are in Appendix 5.3.

Practical Considerations

The above method for dealing with the horizon problem is idealistic: its processing overhead may not be worthwhile in situations where the search corresponding to the selected output value is already *deep enough*.

For example, suppose we have two possible output values – “0” with a probability of 0.5001 and “1” with a probability of “0.4999 – and “1” has the higher score. The above method would require us to repeat the extension of the search for “1” from the current node with an effective probability of 0.5001, but this is not much improvement on 0.4999 for a significant amount of computation.

If the probability of the selected output is *close enough* to $\text{Probability}_{\text{Max}}$ we may choose *not* to perform a secondary search.

The approaches to the horizon problem in this article may be pragmatically ignored when the horizon problem is not serious enough to merit the processing to resolve it.

Resolving the Horizon Problem at Input Event Nodes

For input event nodes the approach will depend on how the score to be assigned to an input node is determined from the backed-up scores corresponding to different input values. Things may be more involved and I will not attempt a complete solution here. The problem, however, is likely to be less severe for inputs: it is probably outputs that we need to worry about. This is because the process of determining the score to be placed at the node is likely to involve some averaging which takes account of probabilities and when a score is unreliable by being backed-up from the path for a low

probability input value then averaging will automatically reduce its influence on the score to be assigned to the node. We may still want to take account of the reliability of scores, however, and how we do this may depend on how the scores associated with different input values are combined.

Input Event Nodes where the Score is the Highest Backed-Up Score

In this situation the backed-up scores are used *optimistically* to determine the score to be placed at an input event node. All the scores backed-up to the input node from the paths corresponding to different input values are compared and the *highest* score assigned to that node. This will probably be too optimistic in most situations.

Scores are handled in such a situation in the same way as scores for output events, so the method of dealing with the horizon problem is the same as that used for output nodes.

Input Event Nodes where the Score is the Lowest Backed-Up Score

In this situation the backed-up scores are used *pessimistically* to determine the score to be placed at an input event node. All the scores backed-up to the input node from the paths corresponding to different input values are compared and the *lowest* score assigned to that node. This may be too pessimistic in many situations.

Scores are handled in such a situation in almost the same way as scores for outputs, except that instead of selecting the highest score we select the *lowest* score. The method of dealing with the horizon problem is the same as that used for output nodes, except that any reference to the *highest* in a set of values is replaced by a reference to the *lowest*.

Input Event Nodes where the Score is the Mean Backed-Up Score

In this situation the aim is neither pessimism nor optimism. The horizon problem may not be important because the probability will be taken account of in calculation of the mean and low probability input values (resulting in shallow searching) will influence the average score less than higher probability input values (resulting in deeper searching). Further measures, however, may be considered necessary, or the situation may be complicated by being some compromise between averaging and optimistic or pessimistic approaches. Such measures are beyond the scope of this article and, given the limited effect of the horizon effect with averaged input scores compared with its effect with output scores, and the way in which the measures needed may depend on specific implementations of the AI system, a description of them is not needed at this stage.

Possible Limitations of the Proposed Solution of the Horizon Problem

The approach to the horizon problem could have limitations due to tendencies which the situation may sometimes have. We need to consider what happens in two kinds of situations:

- Sometimes the short-term effects of most actions, including the best ones, may be generally better than long-term ones. In such situations, higher probability outputs, with the searching associated with them being extended further, have an automatic advantage. A lower

probability output may be better in the long-term, but its long-term consequences will not be explored. This could cause some potentially useful behaviour to be missed, but is not disastrous. The favoured behaviour is what the modelling system considers the most probable anyway – meaning it is likely to be the best available behaviour. Although technically a limitation, this is only an extreme example of what planning as modelling does anyway. We cannot search all the set of possible futures and must occasionally miss opportunities.

- In other situations the short-term effects of most actions, including the best ones, may be generally worse than long-term ones. In such situations the lower probability outputs, with the searching associated with them cutting off early, have an initial advantage over higher probability outputs with deeper searching. This will not cause incorrect selection of outputs, because the approach described here, or something similar, will cause secondary, deeper searching for such outputs. If, though, there is a situation in which there is a tendency for short-term searches to give better results then this secondary searching could occur continually, negating the advantage of planning as modelling. Most situations would not be like this, but if this were a problem we could deal with it. One method could be to return separate shallow and deep search scores for likelier output values – maybe even a number of scores – and only even to consider a lower probability output if its score is better than scores returned by *shallow* searching for likelier output values.

Possible Later Refinements

Output Vectoring System Adjustment

Planning as modelling may later feature some kind of adjustment of the output vectoring system. The output vectoring system currently prioritizes sequences of future input events and output events according to probability. This is prejudiced in favour of outputs which lead to likely futures. If the modelling system were not being used to constrain the search then it would probably need to be restricted to sequences containing a certain number of future input and events: any search tree being used would be a flat one. It may be that output vectoring would work better with less constraint from the modelling system, so that the system's behaviour is a compromise between the output vectoring as described here and a "flat", unconstrained search, making the system's behaviour like that of the current system with all of the probability values from the modelling system "nudged" towards randomness (e.g. 0.5 in a system with binary inputs and outputs) to some degree (although there are other ways the degree of constraint could be moderated). It could, however, be preferable to *exaggerate* the constraint from the output vectoring system in some way. Another approach could be to adjust the scores backed-up the search tree depending on how far down the tree they have been backed-up from. One way to do this would be to apply an adjustment to each score as it is produced at a terminal node, dependent on the node's depth.

Efficiency and Real-Time AI

A range of techniques may be used to increase the efficiency of the output vectoring system's search. Techniques for increasing the efficiency of tree searches are well known, many have been found in researching chess algorithms [10], and will not be discussed here in detail.

Although a tree search is used in this article, and that is how I usually think of it, planning as modelling does not demand a tree search: it merely demands that the output vectoring system perform a search for optimum behaviour constrained by future predictions of inputs and outputs.

Efficiency and management of resources may be particularly important for an AI system in a real-time environment. The system described here treats reality like a chess game played without a clock, assuming that the world will wait while the system processes each input or output, before delivering the next input or accepting the next output. In reality, the system may need to limit its processing to keep within time constraints. The depth of the output vectoring system's tree search, if a tree search is used, may need to be controlled by varying the cut-off probability in accordance with available time. One simple way of achieving this could be to search repeatedly, increasing the cut-off probability each time, until the system runs out of time and is forced to make an output based on what its last search found. A practice of performing repeated searches, increasing the search depth each time, is already used in some chess programs.

An obvious way of increasing efficiency would be to partially reuse the results obtained by the output vectoring system's search for one output to reduce the computation needed for a slightly later output. This makes sense because if there is an output A and a slightly later output B then part of the future examined during the output vectoring system's search for Output A will be part of the future which follows the making of Output B. The idea of using results obtained from the planning of one move to reduce the computation involved in planning of a later move is already known in chess programming. There is the complication, when doing this with planning as modelling, that some of the decisions about backing up scores will be based on future input and output event probabilities which will have changed by the time the later output is performed but this could be dealt with.

An obvious, though not necessarily correct, way of resolving real-time issues may be to use some of the prioritization control outputs to "tune" the output vectoring system, as well as controlling prioritization in the modelling system, so that the modelling system itself has some role in deciding how the output vectoring system is used. Another possibility is that there may be some demonstrably correct adjustment to make, or that it is not needed at all.

Use of a Directed History

The system's response to inputs is based on modelling of previous inputs and outputs. There needs to be a history of desirable behaviour to establish a pattern of desirable behaviour which modelling will continue or, which is better, a history of *improvement* in behaviour to establish a pattern of improvement in behaviour which modelling will continue.

The output vectoring system deals with this by finding ways to improve the system's predicted behaviour slightly, so that this new behaviour can be used instead and will then itself contribute to the pattern of previous behaviour. The output system therefore establishes the pattern of improvement in behaviour for modelling to continue.

An alternative way of establishing a pattern of desirable behaviour improvement in behaviour could be to set up an artificial pattern of previous behaviour as a *directed history*. This could be done in a

number of ways. One way would be to directly program the information. Another way would be for a human to control the system for some time and “act” the part of a system which is exhibiting desirable behaviour or, preferably, an improvement in behaviour. I am unsure about how often this would be done: it is not really part of the main idea, but needed mentioning as a possibility. It would probably be done more often for simpler AI systems where a specific type of desirable behaviour, rather than any improvement, was required or for AI systems with specific behavioural requirements.

Comparison with the Hawkins System

Planning as modelling breaks the traditional partition between planning and modelling. This partition still applies in the hierarchical system of meaning extraction proposed by Jeff Hawkins [11,12] which features a special output hierarchy, “closely-coupled” with the meaning extraction hierarchy. Such partition of planning and modelling is redundant in the planning as modelling approach. If a hierarchical approach were used for the modelling system in planning as modelling then the same hierarchical processing would model reality and plan actions.

Philosophical Issues

Planning as modelling, if considered as a proposal for human decision making, has philosophical implications:

The “System X” Analogy

One way of viewing planning as modelling is as follows:

Imagine that you observe the behaviour of an artificially intelligent machine. You imagine that the machine is running some software called “System X” and you make a model to predict System X’s behaviour. The machine itself, however, is doing the same thing. It is also making a model to predict the behaviour of “System X” and using it to *generate* its behaviour.

In a way, this could be taken as meaning that System X does not really exist! Even the computer which would be running System X is just modelling it and the mind, already arguably a virtual thing, can be viewed as a virtual shadow of itself. The artist, Rene Magritte’s, work “La trahison des images” (“The treachery of images”) comes to mind here.

“Free Will”

Many people believe in “free will”. This is generally understood to mean that humans are not “forced” to act in a certain way by some mechanism, such as the physical structures of brains, but are “free” to act as they want. I go further than thinking that free will, defined in this way, does not exist: I think it is an incoherent concept. I could be argued with on this – maybe about the definition of free will, though I think that the way I described is how it is generally considered. Why do people feel they have “free will”?

Planning as modelling suggests a reason for this feeling. No decision making system can know with certainty what its future decisions will be until it has made them because becoming aware of what a future decision is going to be is equivalent to making the decision. For example, it is incoherent to say, “I am 100% certain that I will *decide* to take an umbrella to work tomorrow.” If you know you will you have already decided. If a system already knows with certainty what the result of some decision is going to be then it can cancel any processing that it was going to do to make the decision and just use its own advance knowledge of the decision *as* the decision: predicting a decision is the same as making it.

Any distinction between *predicting* decisions and *making* them is meaningless: it is all the same process and any indeterminacy of future decisions is equivalent to “not having decided yet”.

Humans tend to feel they have free will but we have no reason to think that other systems, such as thermostats or current chess playing programs have anything resembling this. We could explain this by saying that these systems are not intelligent enough – and it would probably be correct – but it does not really deal with the issue. If a system became smarter would that in itself mean that it would acquire free will? Is our feeling of free will caused by general intelligence and decision making per se, or something specific (which may of course be required for general intelligence)?

In planning as modelling the system probabilistically predicts not just its future inputs, but its own outputs – its own behaviour. A system using planning as modelling is facing the very indeterminacy that I discussed above. Its own behaviour is just another part of reality that, like the rest of reality, it can only model statistically. From the perspective of the system, the system itself appears to be something beyond the control of the system because it is trying to predict what it will do and not being completely successful. It may seem that the system could improve its prediction of some future output, decreasing uncertainty, but this improvement in prediction is merely part of the *decision* itself. No matter how quickly the system examines a future decision to improve its prediction of it, it can never be quick enough to precede the decision itself because the very improvement of the prediction *is* the decision. The system is in a situation rather like that of a child trying to open a refrigerator door quickly enough to see inside before the light turns on. If humans use something like planning as modelling this “refrigerator light” situation could cause the perception of indeterminacy of our actions and the feeling of free will.

Conclusion

This article has combined the earlier discussions of planning as modelling [1,2,3,4,5,6] (see Appendix 6). Planning as modelling uses a modelling system which makes probabilistic predictions to plan the future behaviour of an AI system. Modelling is made equivalent to planning. An important aspect of this is that the modelling system itself makes no distinction between the AI system’s inputs and outputs: it observes both and makes predictions for both. This can be summarized as follows:

Planning is modelling.

This is better than the system suggested by Hawkins [11,12], in which planning is still a discrete process, even if closely linked to modelling.

In planning as modelling, a *modelling system* observes the AI system's inputs and outputs and makes probabilistic predictions of future inputs and outputs. These are used to constrain a tree search by the output vectoring system for optimum behaviour, the search tree being cut off when the future described by a given node falls below a certain level of probability. Special pseudo-outputs, known as *prioritization control outputs*, are generated by the AI system in the same way as conventional outputs, but instead of acting on the outside world they act on the modelling system as instructions to control its prioritization of computing resources. Because prioritization control outputs are generated in the same way as other outputs then no special learning process is needed. The system will learn to control its own modelling system in the same way as it learns any other behaviour. In a way, the modelling system, with respect to outputs, is treated as part of the external world, which the system must learn to manipulate, as discussed previously [5]. Improvements in prioritization control will make the modelling system more effective, in turn improving prioritization control further, so prioritization control outputs are important in allowing the AI system's learning.

Although this article has discussed planning as modelling using a tree search and cut-off probability, the output vectoring system's search does not have to be a tree search. It can be any search of possible futures in which futures are prioritized for examination according to their probability of occurrence according to the modelling system. If a tree search is used then the method of applying constraint from the modelling system does not have to be based on the running probability and cut-off probability. It can be based on different methods – providing that they cause the search to cut-off when nodes are reached by traversing many branches of the search tree corresponding to input and output events with unlikely values.

Example program code for basic AI systems using planning as modelling in Standard Pascal and Object Pascal/The Delphi Programming Language has been provided in Appendix 3. These programs show how the modelling system can be used to implement planning as modelling, but omit implementation of the modelling system. Methods/subroutines to provide the interface between the modelling system and the remainder of the AI system are, however, provided. The main purpose of the program listings is to demonstrate, unequivocally, one way in which planning as modelling could work.

Although a separate system, the *output vectoring system*, is still needed outside the modelling system this is better than having a planning system outside the modelling system because the output vectoring system is not doing anything profound and its implementation is more software engineering than real AI research. Modular design is possible for the modelling system and output vectoring systems. The output vectoring system does not need to “know” how the modelling system works. Output vectoring system designs are not specific to models and reusable output vectoring systems can be produced. This allows successive, model-independent output vectoring systems to be made, each more efficient than the last as knowledge is gained about increasing their efficiency, making very efficient output vectoring systems a realistic prospect. Likewise, a modelling system can be designed without knowledge of how the output vectoring system will work.

Adjustment of the output vectoring system may be desirable to reduce the amount of constraint provided by the modelling system, making the system's behaviour a compromise between the sort of search process described here and a "flat" search. It may, however, be preferable to exaggerate the constraint provided by the output vectoring system instead.

Issues of efficiency and real-time processing need resolving, but are likely to be conventional software engineering issues of secondary importance. The main requirement now to make this work as a simple AI system is a suitable probabilistic modelling system. How good this modelling system is will determine how well the AI system works. Prioritization control outputs are an important concept here, and although a modelling system could be made which does not use them, they would be needed for it to work very well.

One way of viewing the system is in terms of the "System X analogy", according to which the AI system is modelling a different system, System X, and thereby becoming equivalent to System X.

Something like planning as modelling in humans could explain the widespread belief in "free will". A system using planning as modelling is attempting to predict its own future decisions, and therefore is confronted by its inability to do this except in probabilistic terms. Although the knowledge about future decisions can be improved, the lack of distinction between planning and modelling means that finding out more about the decision is equivalent to making it, so knowledge can never precede a decision.

References

- [1] Web Reference: Almond, P. (2006). *Planning as Modelling in AI*. Retrieved 26 November 2006 from <http://www.paul-almond.com/PlanningAsModellingInAI.pdf>.
- [2] Web Reference: Almond, P. (2006). *Programming of Planning as Modelling in AI*. Retrieved 28 December 2006 from <http://www.paul-almond.com/ProgrammingOfPlanningAsModellingInAI.pdf>.
- [3] Web Reference: Almond, P. (2006). *How AI Would Work*. Retrieved 4 September 2006 from <http://www.paul-almond.com/HowAIWouldWork.pdf>.
- [4] Web Reference: Almond, P. (2006). *Occam's Razor Part 9: Representation and Planning of Actions in Artificial Intelligence*. Retrieved 29 July 2006 from <http://www.paul-almond.com/OccamsRazorPart09.pdf>.
- [5] Web Reference: Almond, P. (2006). *AI as a Boundary System*. Retrieved 17 September 2006 from <http://www.paul-almond.com/AIAsABoundarySystem.pdf>.
- [6] Web Reference: Almond, P. (2007). *Resolving the Horizon Problem in Planning As Modelling*. Retrieved 30 March 2007 from <http://www.paul-almond.com/ResolvingHorizonProblem.pdf>.
- [7] Levy, D.N.L. (1984). *The Chess Computer Handbook*. London: Batsford. Chapter 3, pp38-52.

[8] Ibid. Chapter 2, pp7-37.

[9] Ibid. Chapter 6, pp82-84.

[10] Heinz, E, A. (2000). *Scalable Search in Computer Chess: Algorithmic Enhancements and Experiments at High Search Depths*. Vieweg Verlag. Chapter 0, pp11-18.

[11] Hawkins, J., Blakeslee, S. (2004). *On Intelligence*. New York: Henry Holt.

[12] Web Reference: George, D., Hawkins, J. (?). *Belief Propagation and Wiring Length Optimization as Organizing Principles for Cortical Microcircuits*. Retrieved 24 April 2006 from <http://www.stanford.edu/~dil/invariance/Download/CorticalCircuits.pdf>.

[13] Web Reference: Almond, P. (2006). *Occam's Razor Part 6: Partial Models as "Envelopes"*. Retrieved 1 March 2006 from <http://www.paul-almond.com/OccamsRazorPart06.htm>.

[14] Web Reference: Almond, P. (2006). *Occam's Razor Part 7: Hierarchy and Ontology*. Retrieved 30 April 2006 from <http://www.paul-almond.com/OccamsRazorPart07.htm>.

[15] Web Reference: Almond, P. (2006). *Occam's Razor Part 8: Modelling in Artificial Intelligence*. Retrieved 9 June 2006 from <http://www.paul-almond.com/OccamsRazorPart08.pdf>.

[16] Web Reference: Almond, P. (2006). *Downward Transfer of Probabilities in AI*. Retrieved 15 October 2006 from <http://www.paul-almond.com/DownwardTransferOfProbabilitiesInAI.pdf>.

Appendices

Appendix 1: Summary of Planning as Modelling

- *Input and output events* occur. Each has a value at some instant of time.
- Modelling system informed about input and output events as they occur and probabilistically predicts the values for future input and output events.
- No distinction between input and output events in the modelling system: both just treated as data on which future predictions are based.
- Internal modelling system workings not specified.
- When an input event occurs the modelling system is informed of the input event and its value.
- When an output event is to occur the *output vectoring system* determines the best output value. The AI system then makes this output and the modelling system is informed of the output event and its value.
- Modelling system has services allowing the output vectoring system to use it to provide constraint in its search.
- Modelling system services based on *current event* – the input or output considered (actually or hypothetically) to be happening *now*. Each time the modelling system is informed of a real or hypothetical input or output event the current event is moved to the next (chronologically) input or output event.
- Output vectoring system searches possible futures using a tree search as in chess programs.
- Each search tree node = an input or output event.
- Each search tree branch = a particular value for an input or output event.
- Search constrained by *running probability*. Initially, running probability = 1. For each branch the modelling system is requested to provide p , the probability of the current event occurring with that particular input or output value. For the next node:
running probability = running probability $\times p$
- After following a branch to the next node, the modelling system is informed that the input or output event has occurred with the corresponding value.
- Branches only extended from a node if running probability $<$ *cut-off probability*.
- If running probability \geq cut-off probability the node becomes a terminal node.
- Situational evaluation function assigns scores to terminal nodes. Does not need to use abstraction in model – can use (most likely, recent) history of input and output events for the model or the information about input and output events may be used to maintain a data structure which the situational evaluation function uses to assess the situation's utility. The data needed for situational evaluation can be maintained and provided by the modelling system or by a separate system.
- Scores backed-up from nodes as in chess search trees.
- At output event nodes the highest backed-up score is assigned to the node and then backed-up to higher nodes.
- At input event nodes the mean backed-up score may be assigned to the node and then backed-up to higher nodes, or the highest or lowest score may be used, or some compromise between the mean score and highest or lowest.

- Any changes to the model, current event and running probability for following branches to lower nodes are only in effect during processing at lower nodes. This could be achieved by making a copy of the model each time processing goes to a lower node, but an undo capability in the modelling system is better. After processing at a lower node is complete this allows the modelling system to be instructed to undo any changes made as a result of being informed about input or output events happening and to revert the current event to the (chronologically) previous event.
- The first node in the output vectoring system's search tree corresponds to the current event at the time the output vectoring is started. The score backed-up to this node and selected at it corresponds to the best output value available at this node and this output value is returned from the output vectoring system as the best output available.
- The best output is made by the AI system and the modelling system is informed that this output event has occurred with the best output as its value.
- Planning as modelling can be understood more generally. The output vectoring system's search need not be a tree search. It can be any search of possible futures in which futures are prioritized for examination according to their probability of occurrence according to the modelling system, a possible future being some hypothetical situation reached by occurrence of a sequence of input and output events with particular values. If a tree search is used then the method of applying constraint from the modelling system does not have to be based on the running probability and cut-off probability. It can be based on different methods – providing that unlikely futures are eliminated – unlikely futures being ones reached via unlikely sequences of input and output events.
- Planning as modelling also uses prioritization control outputs – special pseudo-outputs generated by the system in the same way as outputs, but instead of acting on the external world, acting on the modelling system as instructions to control prioritization of its use of computing resources.
- A horizon problem can occur because an unlikely future could occasionally return the highest score.
- For output event nodes, one horizon problem solution is to repeatedly extend a secondary search from the current node for the output value with the highest score backed-up to a node, but deepening the search by using the probability for the highest probability output as the *effective probability* for that output value, replacing the score obtained for this output value by this “deeper search” score, until the highest scoring output value has been obtained by using the same effective probability as the most likely output value.
- For input nodes, the horizon problem is likely to be less severe. The method of dealing with the horizon problem depends on how the score to be assigned to an input node is selected from backed-up scores. If the score assigned to the input node is the *lowest* of the backed-up scores then this is treated in the same way as an output event node, except considering the *lowest* scores as if they were the *highest* scores. If the score assigned to the input node is the *highest* of the backed-up scores (an optimistic approach not likely to be used often) then this is treated in the same way as an output event node. If the score assigned to an input node is the mean (average) of the backed-up scores, or some more complicated approach is used then a different approach to the horizon problem may be needed, though the horizon problem should be less severe.

Appendix 2: Interaction with the Modelling System

This appendix gives an example set of services that a modelling system could provide to allow it to be used to provide constraint to an output vectoring system.

The services here are used in the example programs in Appendix 3. Subroutine names in the example programs are given in *italics*.

Interaction as Services

We only need to deal with input and output events singly because they are dealt with in chronological order:

- When the modelling system is being informed about the results of real input or output events these occur in chronological order as the real input or output events happen.
- When the modelling system is being informed about the results of hypothetical real input or output events these updates will usually occur in chronological order as they do, for example, if a tree search approach is used: each branching of the search tree corresponds to the next (in a temporal sense) occurrence of an input or output.

The modelling system does not need to store the details of all the input events and output events that it has been informed about explicitly: it could store some abstraction of many input and output events internally, or it may determine that a particular input event or output event is irrelevant and disregard it completely. None of this matters outside the modelling system. Once the modelling system has been informed that a particular input event or output event has occurred with some value then it can be assumed that the modelling system's future predictions will take account of this input or output event, or of some abstraction of this event and others, or ignore it completely if it is irrelevant: all this is specific to the modelling system.

Not needing to do more than process inputs and outputs in chronological order eliminates some complexity in the interaction of the modelling system with the rest of the AI system. All we need consider is the input or output that is happening *now* – whether this is a real “now” for the system's real inputs and outputs or a hypothetical “now” reached at some point in the output vectoring system's search. The modelling system needs a way of telling us about the probabilities associated with this input or output event. We also need some way of moving from one event to the next chronologically and a way of making hypothetical inputs and outputs in the output vectoring system's search. This can be provided by a simple set of services, based on the idea of the *current event*.

The Current Event

Input events and output events occur in chronological order. At any time the modelling system is considering a particular input or output event as being the *current event*. The current event is not necessarily the input or output event that is happening now in the real world. The current event is the input or output event for which the following apply:

- The modelling system needs to make probabilistic predictions. Any probabilistic prediction that it is requested to provide for an input or output event will be about the current event.
- The modelling system needs to modify itself to take account of input or output events that have happened in reality or are being hypothetically regarded as having happened by the output vectoring system. If the modelling system is informed that a particular input or output event has happened with a particular value (e.g. that “0” or “1” has been input or output) then the input or output event about which it is being informed is the current event.

Changing the Current Event

The current event will need to change for two main reasons:

- Occurrence of real input and output events – As time passes, input and output events occur in reality. The modelling system needs to be told what value each of these has, so each becomes, in turn, the current event.
- Occurrence of hypothetical input and output events – As the output vectoring system simulates possible futures it needs to simulate possible future input and output events. If a tree search is used, for example, an input or output event would be simulated at each node of the search tree, just as each node of a chess search tree simulates a move. Each of these simulated, or hypothetical, input and output events becomes, in turn, the current event.

Both of these situations – the real and the hypothetical – involve input and output events being observed by the modelling system, meaning the modelling system is informed about them, in chronological order. This means that when the current event is changed it will always change to the *next* input or output event (with one exception, relating to “undoing” in the recursive process, that will be discussed shortly). There is no need for any complex system for telling the modelling system *which* input or output event is the next one. The modelling system merely needs to provide a service (*NextInputOutputEvent*) which allows it to be instructed to make the *next* input event or output event (chronologically) the current event.

Updating the Model

The modelling system observes the AI system’s input and output events and makes probabilistic predictions of future inputs and outputs. An observation of an input or output involves the modelling system being informed of the value that was input or output for the input or output event that is happening, really or hypothetically, *now*. After being told this the modelling system modifies itself so that the predictions that it will produce for other, future, input and output events are altered accordingly. We do not need to know how this works: it is only an issue in the design of the modelling system itself.

For this to happen the modelling system needs a single service (*ApplyInputOutputEventToModel*) that allows it to be told what the value of the input received or the output made is for the current event so that it can update itself accordingly. A second service (*DoInputOutputEvent*) is actually provided which combines this with prioritization control (see below).

Obtaining Information About the Current Event

Within the output vectoring system, probabilistic predictions need to be obtained from the model for future input or output events. This requires the modelling system to have a service that, given some particular input or output value, gives the probability that the current input or output event will be found to have that value. For example, we may want to know the probability that the current input or output event has a value of “0” or we may want to know the probability that it has a value of “1”.

If the inputs and outputs are all binary then a service which just returned the probability of the “1” input or output would be adequate – the probability for “0” being obtained by subtracting it from 1, but a more general approach should work for different input and output values. In the example programs in Appendix 3, binary inputs and outputs are used, but I have defined a service (*EventProbability*) that returns the probability of any given input or output or output value – although it is only used for “0” and “1” input and output values and therefore accepts a bit as a parameter.

As well as probabilities, other information needs to be obtained about the current input or output event:

- A service (*IsOutputEvent*) is needed to indicate whether the current event is an input event or output event. Another service (*IsInputEvent*) is also provided for completeness.
- Some outputs are special prioritization control outputs, meaning that they are used to control prioritization of computing resources in the model itself, rather than being sent to the outside world. A service (*IsPrioritizationControl*) is needed to indicate, in situations where the current event is an output event, whether or not it is a prioritization control output event.

Undoing

When the output vectoring system is investigating possible future inputs and outputs the model is put into corresponding states reached by informing it about various sequences of hypothetical, future inputs and outputs. These updates are only wanted while performing that part of the search and need to be undone later.

The modelling system needs a service to allow undoing of changes that have recently been made to it. I define two services here:

- A service (*UndoApplyInputOutputToModel*) which undoes the most recent informing of the modelling system about the value for the current input or output event which has not already been undone.
- A service (*UndoNextInputOutputEvent*) which undoes the most recent changing of the current event which has not already been undone – equivalent to making the event before it (chronologically) the current event again.

An alternative to an undo facility would be to make a *copy* of the modelling system each time it needed updating, and then when this copy needed updating to make a copy of it and so on. “Going back” would now just mean discarding the copy currently in use and reverting to the previous version of the model. This would be like a chess tree search passing a copy of the current state of the chess game to the next level of recursion as a value parameter. While this would work in principle, it could be wasteful of computing resources and would be questionable, but anyone wanting to use the modelling system in this way could do so whether an undo facility was included or not: nothing is gained by omitting an undo facility.

Evaluating

Planning as modelling needs an evaluation function. I define this as a service (*SituationalEvaluationFunction*) of the modelling system, which works in a similar way to situational evaluation functions in chess algorithms [8]. It gives a score indicating the desirability of the model in any given state. The situational evaluation function does not need to use any abstraction produced by the modelling system but can simply use the (most likely, recent) history of input (and possibly, output) events, or the modelling system may use information about input and output events to maintain a data structure for the situational evaluation function to use.

Inputs and Outputs

Services are needed to allow the AI system to interact with the outside world.

- A service (*GetActualInput*) obtains the next input due from the outside world. This should relate to the current event.
- A service (*MakeActualOutput*) sends an output to the outside world with some specific value. This should also relate to the current event.

For now, I have included these services in the modelling system, even though they are not, technically, modelling activities, because they rely on information specific to the model.

Applying Prioritization Control Outputs

Planning as modelling uses prioritization control outputs – special pseudo-outputs that are sent back into the AI system and act as instructions to control prioritization of computing resources in the modelling system. Normal outputs are only applied when the system is responding to real

inputs and are not applied when the output vectoring system is simulating possible futures, while prioritization control outputs, when they occur, are always applied. Furthermore, the effects of prioritization control outputs need to be undone when they are used hypothetically while the idea of undoing real outputs makes no sense. Prioritization control outputs are therefore not dealt with as normal application of outputs in the software provided here.

Prioritization control outputs are instead dealt with separately by another service (*ApplyPrioritizationControlToModel*). It is convenient to apply prioritization control outputs at the same time as updating the model because both updates of the model and prioritization control involve changes to the model that are performed whether the situation is a hypothetical one or not. Both prioritization control outputs and updates to the model may need to be undone. Because there is such a close link between model updates and application of prioritization control outputs, I have also provided a service (*DoInputOutputEvent*) in the software which combines updating of the model with application of prioritization control outputs so that a single command can be used to inform the model of an output event occurring with a particular value and to apply that output to the model as a prioritization control output.

An alternative approach could have been used in software design: I could have dealt with the prioritization control outputs as very similar to real outputs (which is how they should conceptually be viewed), so that the service that makes outputs would also apply a prioritization control output, while only actually making the real output if the situation is not hypothetical.

Appendix 3: Example Computer Programs

Appendix 3.1: Comments on Example Computer Programs

Two programs to demonstrate planning as modelling are in Appendices 3.3 and 3.4.

These programs cannot be used in their current form because software is not provided to implement the modelling system. Planning as modelling does not specify how to make a modelling system, but how a modelling system making probabilistic predictions can be used to plan the actions of an AI system, eliminating the need for a separate planning system. The computer code in Appendices 3.3 and 3.4 therefore assumes that a working, probabilistic model is available. Although the implementation of the modelling system is omitted, subroutines are given to act as an interface between the modelling system and the program using it. What is provided is what would be a basic, working AI system *if* the probabilistic modelling system were added. No attempt to resolve the horizon problem is made in these programs.

The program listings may appear short, given that they are supposed to provide the basis on which an AI system can work. They would be even shorter if certain conventions had not been followed and they had not been structured to try to make them easy to understand and analyse. The programs should be expected to be short, however. All that intelligence does is model and plan. Planning as modelling suggests that planning can be done in the modelling system, so it is appropriate that there should be hardly anything outside the modelling system. Ideally there would be *nothing* outside it, but this is not completely attainable in practice: after all, without using any systems outside of the modelling system there is nothing we can actually do to implement planning as modelling, but we can get very close.

One example program is in Standard Pascal and one in Object Pascal/The Delphi Programming Language (the version of Pascal used by Borland's Delphi Pascal compiler). The programs contain some sections that are commented "{The code here is specific to the modelling system.}" where code would need to be placed to add the functionality of a specific modelling system and provide a working AI system. These programs might be considered a proposed framework for an AI system.

The example programs assume that each input or output of the system is a single binary digit (bit), but this does not have to be the case. The software could easily be modified to have any number of branches from each search tree node – one for each possible input or output value. The example programs also use a tree search in the output vectoring system. Planning as modelling does not require a tree search, but it seems a sensible approach and I generally imagine it like this.

Standard Pascal Program in Appendix 3.3

The first example, in Appendix 3.3, is written in Standard Pascal. This program has the main control logic, modelling system and output vectoring system in a single source code file. The emphasis is on staying within Standard Pascal rather than good structuring of the system so that the program can be understood without knowledge of any specific variation of Pascal.

The model's state needs to be preserved throughout a run of the program: modelling system subroutines change a model which then remains in this state for other modelling system subroutines to extract information from it until later modelling subroutine calls make further changes to it. This makes Standard Pascal less than ideal for this purpose: variables in a Standard Pascal subroutine cannot maintain their states between different calls of the subroutine. There are solutions to this and, as the implementation of the modelling system is not shown here, no position is taken about which is used. One solution is for the model itself to be maintained in a separate program with which the modelling system subroutines in this example interact. Another is for the model data to be maintained in a file. The most obvious solution, and an easy one to implement, is for the model to be represented by global variables. Global variables, however, are generally met with disapproval, as all of these methods will be by some readers.

Object Pascal/Delphi Programming Language in Appendix 3.4

Even when implemented in Standard Pascal, the system design essentially involves objects: the approach is already treating the model as an *object* with which there is interaction only by means of certain subroutines. An obvious approach is to use a variation of Pascal which allows explicit description of the model as something that maintains its state.

The second example, in Appendix 3.4, is written in the version of Pascal used by Borland's Delphi Pascal compiler. This is officially called *The Delphi Programming Language* by Borland but is sometimes known as *Object Pascal*. Object Pascal/The Delphi Programming Language provides object oriented programming facilities, which are used in this program to improve structure.

This second version of the program consists of three parts:

- **Main Program** – coordinates interaction of the modelling system and output vectoring system and uses the modelling system and output vectoring system units (below).
- **Modelling System Unit** – a separate unit providing the modelling system. The model is declared as an object class *TModel*, so when the main program needs to use a model it declares an instance of this class. Currently, only one such model is expected to be needed. The modelling system also declares methods for the model class to allow the model to be changed and information to be extracted from it from outside. This unit is used in the main program and the output vectoring system (see below).
- **Output Vectoring System Unit** – a separate unit providing the output vectoring system. The output vectoring system mainly consists of subroutines that perform a recursive tree search so it may seem that there is little need to declare the output vectoring system as an object class: the unit could just contain the subroutines. An output vectoring system object class *TOutputVectoringSystem* is declared, however, because there is one piece of data that is stored for the output vectoring system – the cut-off probability, which can be set from outside by using an output vectoring system method (*SetCutOffProbability*). Also, later developments in the output vectoring system may involve storage of more information from one instance of use to compute the value for an output to the next, to make the system more suitable for real-time use.

Appendix 3.2: Components of the AI System

Functioning of the Main AI System

The main AI system coordinates interaction between the modelling system and output vectoring system.

The main program starts by setting up the model in its initial state – in the Standard Pascal program by calling subroutine *InitializeModel*, in the Object Pascal/Delphi Programming Language program by using the *TModel* object's constructor method *Create*.

The main program is about to process real input and output events and undoing any of this makes no sense. After initializing the model, the program uses the modelling system's *DisableUndo* method/subroutine to indicate to the model that the facility to undo updates to the model is not required.

The program then processes each real input or output event in chronological order, as they actually occur in reality. The program repeatedly steps through the real input and output events using the modelling system's *NextInputOutputEvent* method/subroutine. Each call of this method/subroutine tells the modelling system that the main program has moved onto the next input event or output event, which should now become the current event.

For each input or output event the program needs to know whether it is an input event or an output event. This is done by using the modelling system's *IsOutputEvent* method/subroutine, which returns true if the current event is an output event and false if it is an input event.

If the current event is an input event then there is an actual input to collect from the outside world. This is done by the modelling system's *GetActualInput* method/subroutine. After the input is acquired the modelling system is informed of what value it had so that it can modify its predictions for future input events and output events accordingly. This "fixing" is done by using the modelling system's *DoInputOutputEvent* method/subroutine.

If the current event is an output event then the AI system needs to decide what output to make. This is the whole reason for use of planning as modelling. The system uses the output vectoring system's main method/subroutine *GetSelectedOutput* to determine the optimum output to make. This returns the optimum output – "0" or "1". This output will need to be made in the outside world, but only if it is not a prioritization control output: prioritization control outputs are special pseudo-outputs that only have an effect on the model itself. The program uses the modelling system's *IsPrioritizationControl* method/subroutine to check if the current event is a prioritization control output and only if this is not the case is the output sent to the outside world using the modelling system's *MakeActualOutput* method/subroutine. An important feature of planning as modelling is that the system's predictions are based on observation of its inputs and *its own outputs*. Therefore, regardless of whether this output is a prioritization control output or not, the modelling system is told what its value is – "0" or "1" – using the modelling system's *DoInputOutputEvent* method/subroutine – just as when dealing with an input. This

method/subroutine will also deal with the application of prioritization control, should this be a prioritization control output.

Notes:

This design does not deal with any real-time issues. It is assumed that the system never misses an input or output event by taking too long processing previous events. One way of imagining this is to assume that all input events are buffered and that there is no time pressure to make any outputs. In many real systems provision would need to be made for real-time requirements, for example by making the depth of the output vectoring system's search process dependent on available time or interrupting it and forcing it to give its best answer. This is an issue which can be resolved by conventional software engineering approaches.

The first action of the system after initializing the model is to disable undo for it. Because undo is disabled in the model's natural state, it would make more sense for the modelling system's *InitializeModel* subroutine or *Create* method to disable undo as part of its initialization of the model. I wanted to make it more obvious in these programs that this was happening and so put it in the main program. Undo is disabled in the main program, and not re-enabled at any point in the main program, so it may seem that undo is never enabled. In fact it is – just at a lower level – inside the output vectoring system's code when *GetSelectedOutput* is running.

Functioning of the Modelling System

The purpose of the modelling system is to analyse real or hypothetical previous inputs and outputs and make probabilistic predictions for future inputs and outputs.

InitializeModel – This subroutine is in the Standard Pascal program only. It sets the model up in its initial state.

Create - the Object Pascal/Delphi Programming Language program's equivalent of the *InitializeModel* subroutine. It is the constructor for the *TModel* object, which is the model. It sets the model up in its initial state.

EnableUndo – enables the undo facility. While it is enabled, any of the modelling system's methods/subroutines that make changes to the model can be reversed. This allows the model to be changed hypothetically to simulate possible futures in the output vectoring system.

DisableUndo – disables the undo facility. While it is disabled, any changes to the model are permanent.

NextInputOutputEvent – makes the next input event or output event (chronologically) after the current event the new current event.

EventProbability – returns the probability that the current event will occur with some given value – i.e. that when this event occurs, the AI system will receive an input with this value or make an output with this value. When the current event is an output event this method/subroutine involves

the AI system in making a prediction about its own behaviour. Previous discussions of planning as modelling just had a single probability value for the input or output bit being “1” provided. This newer approach is better as it allows the software to be more easily altered later if inputs and outputs are needed that are not just single bits.

IsOutputEvent – returns true if and only if the current event is an output event. Both conventional outputs and prioritization control outputs will cause this method/subroutine to return true.

IsPrioritizationControl – returns true if and only if the current event is a prioritization control output event. Returns false if the current event is a conventional output event (i.e. not relating to prioritization control) or an input event.

UndoNextInputOutputEvent – undoes the most recent use of the *NextInputOutputEvent* method/subroutine which has not already been undone. i.e. makes the previous input event or output event (chronologically) the current event. Only valid when undo is enabled and when undo was enabled during the original use of *NextInputOutputEvent*.

ApplyInputOutputEventToModel – used to tell the modelling system that an input or output event has occurred and that the relevant input or output value should now be “fixed”. The event to which it relates is the current event, so this method/subroutine tells the modelling system what the input or output value for the current event is. The modelling system updates itself so that any predictions for future input events or output events are altered accordingly. This method/subroutine can be used hypothetically, as its results can be reversed later (see below).

UndoApplyInputOutputEventToModel – undoes the most recent use of the *ApplyInputOutputEventToModel* method/subroutine which has not already been undone. Only valid when undo is enabled and when undo was enabled during the original use of *ApplyInputEventToModel*.

ApplyPrioritizationControlToModel – applies the current output to the model as a prioritization control output with a known value, i.e. the current event, with a given value is used as a command to control prioritization of computational resources in the modelling system. Only valid when the current event is a prioritization control output.

UndoApplyPrioritizationControlToModel – undoes the most recent use of the *ApplyPrioritizationControlToModel* method/subroutine which has not already been undone. Only valid when undo is enabled and was enabled during the original use of *ApplyPrioritizationControlToModel*.

SituationalEvaluationFunction – returns a score indicating the desirability of the situation described by the model. Intended to be used for hypothetical situations described by the model in the output vectoring system’s search of possible futures.

GetActualInput – retrieves the input from the outside world corresponding to the current event. Only valid when the current event is an input event. It is assumed that the next available input from

the outside world *does* relate to the current event: the modelling system's *NextInputOutputEvent* method/subroutine should ensure this.

MakeActualOutput – sends an output to the outside world corresponding to the current event. Only valid when the current event is an output event. It is assumed that the next expected output to the outside world *does* relate to the current event: the modelling system's *NextInputOutputEvent* method/subroutine should ensure this. Prioritization control is not associated with this method/subroutine for reasons discussed earlier.

Extra Services:

The following services are not needed for the modelling system to be used, as their effects can be duplicated by using other services. They can, however, make the modelling system easier to use and are provided for convenience.

UndoEnabled – returns true if and only if the undo facility is enabled. Potentially useful in debugging.

IsInputEvent – returns true if and only if the current event is an input event. The opposite of *IsOutputEvent*.

DoInputOutputEvent – combines the functions of *ApplyInputOutputEventToModel* and *ApplyPrioritizationControlToModel*. It informs the modelling system that the input event or output event has occurred with that value and that the model should update itself accordingly. It also applies the current event to the model as a prioritization control output if the current event happens to be a prioritization control output. This method/subroutine can be used hypothetically as its effects can be undone. Although this method/subroutine is not required in the modelling system it is convenient and is used in the example programs. For reasons discussed earlier, this method/subroutine associates prioritization control with model updating rather than, as you might reasonably expect, with the making of real outputs.

UndoInputOutputEvent – combines the functions of the *UndoApplyInputOutputEventToModel* and *UndoApplyPrioritizationControlToModel* method/subroutine. Undoes the most recent use of *DoInputOutputEvent* which has not already been undone. Only valid when undo is enabled and was enabled during the original use of *DoInputOutputEvent*. Although this method/subroutine is not required in the modelling system it is convenient and is used in the example programs.

Functioning of the Output Vectoring System

The purpose of the output vectoring system is to use predictions of future inputs and outputs made by the modelling system to determine the optimum output value – in this case “0” or “1” – for a particular output event. The output vectoring system is therefore used only when there is an output to make.

The output vectoring system contains just the following methods/subroutines that are invoked from outside it:

GetSelectedOutput – the main method/subroutine in the output vectoring system. It is called when the current event is an output event and the AI system needs a decision about which output to make – in this case “0” or “1”. Planning as modelling involves running simulations of possible future sequences of input events and output events, and the hypothetical changes to the model need to be undone. This method/subroutine therefore first enables the undo facility by using the modelling system’s *EnableUndo* method/subroutine. It starts the AI system’s planning as modelling recursive tree search with a running probability of 1 by calling subroutine *TreeSearch*. This passes the optimum output back up so that it can be returned to the main program. Finally, undo is disabled with the modelling system’s *DisableUndo* method/subroutine. The search process started by this method/subroutine is selectively limited in depth by the cut-off probability (see below).

SetCutOffProbability – only provided in the Object Pascal/Delphi Programming Language program. The Standard Pascal program just uses a constant for the cut-off probability. Sets the cut-off probability to some given value. The cut-off probability is used to control prioritization of searches in the output vectoring system. The value should be higher when computing resources and time available to make decisions about outputs are higher.

GetCutOffProbability – only provided in the Object Pascal/Delphi Programming Language program. Returns the current value of the cut-off probability set by the *SetCutOffProbability* method (above). This is not an essential method and is not used in the example programs but is provided because it may be convenient for debugging.

Output Vectoring System Internal Subroutines:

The following subroutines are internal to the output vectoring system. They are not used from outside it and are called, directly or indirectly, from the *GetSelectedOutput* method/subroutine.

TreeSearch – the subroutine that does the real work of planning as modelling. It is called for each new branch in the planning as modelling search tree. Each new branch corresponds to a decision about the value for a future input event or output event – in this case “0” or “1”. Each time the tree search is called, a probability value – the running probability – is passed to it as a parameter. The running probability indicates the probability that the sequence of input events and output events described by the path through the search tree to the current branch will happen. The tree search can only continue as long as the running probability does not exceed the cut-off probability. The tree search starts two new branches of the search tree from the current node, each corresponding to a different value – “0” or “1” – for the current event. Each new branch is started by a separate call to the *GetLowerBranchScore* subroutine. Each time a new branch is started – for a particular value of some input or output event – the running probability is multiplied by the probability of that input or output event occurring with that value – obtained from the modelling system using the *EventProbability* method/subroutine. In this way the running probability decreases as the tree gets deeper, its value depending on how likely the future is described by the particular path through the search tree. When a particular path through the search tree is stopped – by the running probability being too low – the desirability of the hypothetical situation described by the model is determined

by applying the modelling system's *SituationalEvaluationFunction* method/subroutine. This score is backed-up the search tree. Each non-terminal node therefore receives two scores "backed up" from nodes below it. These need to be turned into a single score to be placed on that node and then passed further back up the tree. If the node is an output event the *GetScoreFromLowerOutputs* subroutine decides what score to place on this node. If the node is an input event the *GetScoreFromLowerInputs* subroutine decides what score to place on this node.

GetLowerBranchScore – causes a single branch of the search tree to be extended from the current node in the *TreeSearch* subroutine, corresponding to a particular value – "0" or "1" – occurring for the input event or output event corresponding to the current node. This subroutine in turn calls *TreeSearch* (see above), resulting in indirect recursion.

GetScoreFromLowerOutputs – decides how to "back-up" scores in the output vectoring system's search tree when the current node corresponds to an output. Each node of the search tree corresponds to an input event or output event and has, in this case, two branches extending from it – for the "0" and "1" cases – to lower nodes, each of which will have a score on it. A single score needs to be "backed-up" to the current node. For an output event the system can be optimistic, because if it ever finds itself in the situation described by this node it can force the more desirable result by making the corresponding output. This subroutine therefore simply backs up the *larger* of the two sub-node scores.

GetScoreFromLowerInputs – fulfils a similar role to the *GetScoreFromLowerOutputs* subroutine (above), except that it backs scores up to an *input* event node. There is more potential for controversy here than for the output node case. I have opted, in this example, to back up the mean (expected) value, taking into account the probability of the branch leading to each sub-node being followed. For some tasks a more pessimistic, less risk-taking approach may be suitable.

Appendix 3.3: Example Program in Standard Pascal

Standard Pascal Program Code

```
program AISystem1 (input, output);
{Illustrates the Planning as Modelling concept for AI.}
{Standard Pascal}
{AISystem1.pas}
{December 2006}
{www.paul-almond.com}
{Copyright Paul Almond 2006. All Rights Reserved}

const
    CutOffProbability = 1E-08; {An example value. If RunningProbability falls
below this value the current branch of the TreeSearch procedure cannot go
deeper.}
type
    TBit = 0..1; {binary digits}
var
    SelectedOutput: TBit; {the optimum output (0 or 1) found by the output
vectoring system for the next output event}

{MODELLING SYSTEM SUBROUTINES}
```

```

procedure InitializeModel;
{Puts the model in its initial state in which no previous inputs and outputs
have occurred.}
{Any probability values associated with future input or output events in the
model will be determined accordingly -}
{probably 0.5.}
begin
    {The code here is specific to the modelling system.}
end; {end of InitializeModel procedure}

procedure EnableUndo;
{Directs the modelling system to store enough information to allow any changes
to it to be reversed.}
{Allows changes to the modelling system by the output vectoring system's tree
search to be hypothetical.}
begin
    {The code here is specific to the modelling system.}
end; {end of EnableUndo procedure}

procedure DisableUndo;
{Directs the modelling system not to store enough information to allow changes
to be reversed and to discard any such information already stored.}
{Used when the model is being affected by real, rather than hypothetical, input
and output events.}
begin
    {The code here is specific to the modelling system.}
end; {end of DisableUndo procedure}

function UndoEnabled: boolean;
{Returns true if undo is enabled, otherwise returns false. Not used in this
program. Included for completeness and because it may be convenient for
debugging in a real system.}
begin
    {The code here is specific to the modelling system.}
end; {end of UndoEnabled function}

procedure NextInputOutputEvent;
{Makes the next input or output event (chronologically) after the current input
or output event the new current input or output event.}
begin
    {The code here is specific to the modelling system.}
end; {end of NextInputOutputEvent procedure}

function EventProbability (InputOutputValue:TBit): real;
{Returns the probability that a value of InputOutputValue (which is 0 or 1)
will be input or output when the current input or output event occurs.}
begin
    {The code here is specific to the modelling system.}
end; {end of EventProbability function}

function IsOutputEvent: boolean;
{Returns true if the current input or output event is an output event,
otherwise returns false.}
begin
    {The code here is specific to the modelling system.}
end; {end of IsOutputEvent function}

```

```

function IsInputEvent: boolean;
{Returns true if the current input or output event is an input event, otherwise
returns false.}
{Always returns the opposite of what IsOutputEvent would return.}
{Not used in this program and only included for completeness.}
begin
    IsInputEvent:= not IsOutputEvent;
end; {end of IsInputEvent function}

function IsPrioritizationControl: boolean;
{Returns true if the current input or output event is a prioritization control
output event (also implying an output event), otherwise returns false.}
begin
    {The code here is specific to the modelling system.}
end; {end of IsPrioritizationControl function}

procedure UndoNextInputOutputEvent;
{Undoes the effects of procedure NextInputOutputEvent, i.e. makes the previous
input or output event (chronologically) before the current input or output
event the new current input or output event.}
{Requires undo to be enabled.}
begin
    {The code here is specific to the modelling system.}
end; {end of UndoNextInputOutputEvent procedure}

procedure ApplyInputOutputEventToModel (InputOutputValue:TBit);
{Puts the model into the state that it should be in after the current input or
output event has occurred with an input or output of value InputOutputValue,
i.e. the relevant input or output bit is fixed.}
{This could be a real or hypothetical updating of the model.}
begin
    {The code here is specific to the modelling system.}
end; {end of ApplyInputOutputEventToModel procedure}

procedure UndoApplyInputOutputEventToModel;
{Undoes the effects of the last use of procedure ApplyInputOutputEventToModel.
Requires undo to be enabled.}
begin
    {The code here is specific to the modelling system.}
end; {end of UndoApplyInputOutputEventToModel procedure}

procedure ApplyPrioritizationControlToModel (OutputValue:TBit);
{Adjusts prioritization in the model, using the current input or output event
as a prioritization control output with value OutputValue.}
{Requires the current input output or output event to be a prioritization
control output.}
begin
    {The code here is specific to the modelling system.}
end; {end of ApplyPrioritizationControlToModel procedure}

procedure UndoApplyPrioritizationControlToModel;
{Undoes the effects of the last use of procedure
ApplyPrioritizationControlToModel.}
{Requires undo to be enabled.}
begin
    {The code here is specific to the modelling system.}

```

```

end; {end of UndoApplyPrioritizationControlToModel procedure}

procedure DoInputOutputEvent (InputOutputValue:TBit);
{Fixes the current input or output event as having occurred, actually or
hypothetically, with value InputOutputValue.}
{Updates the model accordingly so that any subsequent probabilities of input or
output events obtained from the model assume that the current input or output
event has occurred with value InputOutputValue.}
{If the input or output event is a prioritization control output then it
applies prioritization to the model.}
{Included for convenience. The effects of this procedure can be produced by
using procedures ApplyPrioritizationControlToModel and
ApplyInputOutputEventToModel.}
begin
    ApplyInputOutputEventToModel (InputOutputValue);
    if IsPrioritizationControl then
        ApplyPrioritizationControlToModel (InputOutputValue)
end; {end of DoInputOutputEvent procedure}

procedure UndoDoInputOutputEvent;
{Undoes the effects of the last use of procedure DoInputOutputEvent.}
{Requires undo to be enabled.}
{Included for convenience. The effects of this procedure can be produced by
using procedures UndoApplyPrioritizationControlToModel and
UndoApplyInputOutputEventToModel.}
begin
    if IsPrioritizationControl then
        UndoApplyPrioritizationControlToModel;
        UndoApplyInputOutputEventToModel
end; {end of UndoDoInputOutputEvent procedure}

function SituationalEvaluationFunction: real;
{Applies the situational evaluation function to the model and returns a score
indicating the desirability of the situation described by the model.}
begin
    {The code here is specific to the modelling system.}
end; {end of SituationalEvaluationFunction function}

{INPUT / OUTPUT SUBROUTINES}

procedure MakeActualOutput (OutputValue:TBit);
{Sends an actual output to the outside world. Has no effect on the model
itself.}
begin
    {The code here is specific to the modelling system.}
end; {end of MakeActualOutput procedure}

function GetActualInput: TBit;
{Receives an actual input from the outside world. Has no effect on the model
itself.}
begin
    {The code here is specific to the modelling system.}
end; {end of GetActualInput function}

{OUTPUT VECTORING SYSTEM SUBROUTINES}

```

```

procedure Treesearch (var SelectedOutput:TBit; var Score:real;
RunningProbability:real); forward;
{Forward declaration needed because of indirect recursion.}

function GetLowerBranchScore (InputOutputValue:TBit; RunningProbability:real):
real;
{Follows a branch of the search tree to a lower node, returning the score that
gets backed up to it.}
var
    NullOutput: TBit; {The selected output is of no interest at this level of
recursion.}
    Score: real; {the score backed up from lower levels of the search tree}
begin
    DoInputOutputEvent (InputOutputValue);
    NextInputOutputEvent;
    TreeSearch (NullOutput, Score, RunningProbability);
    UndoNextINputOutputEvent;
    UndoDoInputOutputEvent;
    GetLowerBranchScore:= Score
end; {end of GetLowerBranchScore function}

procedure GetScoreFromLowerOutputs (LowerScore0,LowerScore1:real; var
Score:real; var SelectedOutput:TBit);
{Decide what the score should be for this node, based on the two scores backed
up from branches to lower nodes for outputs of 0 and 1.}
{Also returns the output (0 or 1) associated with the selected score, though
this is only of interest in the top level of the tree search.}
begin
    {Select the higher score and the relevant output.}
    if LowerScore0 > LowerScore1 then
        begin
            Score:= LowerScore0;
            SelectedOutput:= 0
        end
    else
        begin
            Score:= LowerScore1;
            SelectedOutput:= 1
        end
    end
end; {end of GetScoreFromLowerOutputs procedure}

function GetScoreFromLowerInputs
(LowerScore0,LowerScore1,Probability0,Probability1:real): real;
{Decides what the score should be for this node, based on the two scores backed
up from lower nodes for inputs of 0 and 1.}
begin
    {Determine the mean (expected) score}
    GetScoreFromLowerInputs:= (LowerScore0*Probability0 +
LowerScore1*Probability1) / 2;
end; {end of GetScoreFromLowerInputs function}

procedure Treesearch (var SelectedOutput:TBit; var Score:real;
RunningProbability:real);
{Searches recursively to determine the better output (0 or 1) for the current
output event.}
var
    LowerScore0, LowerScore1: real; {Scores backed up from the level below.}

```

```

begin
    {Has the cut-off limit been reached yet?}
    if RunningProbability < CutOffProbability then
    begin
        {Branch off for the 0 and 1 cases.}
        {Branch off for the 0 case, decreasing RunningProbability.}
        LowerScore0:= GetLowerBranchScore (0,
RunningProbability*EventProbability(0));
        {Branch off for the 1 case, decreasing RunningProbability.}
        LowerScore1:= GetLowerBranchScore (1,
RunningProbability*EventProbability(1));
        {Decide how to back up the score.}
        if IsOutputEvent then
            GetScoreFromLowerOutputs (LowerScore0, LowerScore1, Score,
SelectedOutput)
        else
            Score:= GetScoreFromLowerInputs (LowerScore0, LowerScore1,
EventProbability(0), EventProbability(1))
        end
    else
        {This is a terminal node. Return a score for it.}
        Score:= SituationalEvaluationFunction
    end; {end of TreeSearch procedure}

function GetSelectedOutput: TBit;
{The main output vectoring system routine. Starts the recursive search to
determine the better output (0 or 1) for the current output event.}
var
    SelectedOutput: TBit;
    NullScore: real; {the score associated with use of this output - not used
at this level of the program but included because the output vectoring system's
routine requires it as a variable parameter}
begin
    EnableUndo;
    TreeSearch (SelectedOutput, NullScore, 1);
    {1 is the running probability which will decrease with successive levels of
recursion depending on how likely the input or output branches are, until the
CutOffProbability threshold is reached.}
    DisableUndo;
    GetSelectedOutput:= SelectedOutput
end; {end of GetSelectedOutput procedure}

{MAIN PROGRAM}

begin
    InitializeModel;
    DisableUndo;
    {The normal state for the system, except when in an output vectoring system
tree search, is for the undo facility to be disabled.}
    {Process inputs and outputs in chronological order.}
    repeat
        NextInputOutputEvent;
        if IsOutputEvent then
        begin
            {Use output vectoring system to determine the better output.}
            SelectedOutput:= GetSelectedOutput;

```

```

        {Do the output for real.}
        if not IsPrioritizationControl then
            MakeActualOutput (SelectedOutput);
        {Update the model.}
        DoInputOutputEvent (SelectedOutput)
    end
    else
        {Get the input value and use it to update the model.}
        DoInputOutputEvent (GetActualInput)
    until false {i.e. just keep running}
end. {end of AISystem1 program}

```

Appendix 3.4: Example Program in Object Pascal (Delphi Programming Language)

Main Program

```

program AISystem2 (input, output);
{Illustrates the Planning as Modelling concept for AI.}
{Object Pascal (Delphi Programming Language)}
{AISystem2.pas}
{December 2006}
{www.paul-almond.com}
{Copyright Paul Almond 2006. All Rights Reserved.}

uses Modelling, OutputVectoring;

const
    CutOffProbability = 1E-08; {An example value. If RunningProbability falls
below this value the current branch of the TreeSearch procedure cannot go
deeper.}

var
    Model: TModel;
    OutputVectoringSystem: TOutputVectoringSystem;
    SelectedOutput: TBit; {the optimum output (0 or 1) found by the output
vectoring system for the next output event}

begin
    {Initialize the modelling system.}
    Model := TModel.Create;
    Model.DisableUndo;
    {The normal state for the system, except when in an output vectoring system
tree search, is for the undo facility to be disabled.}
    {Initialize the output vectoring system.}
    OutputVectoringSystem := TOutputVectoringSystem.Create;
    OutputVectoringSystem.SetCutOffProbability (CutOffProbability);
    {If the running probability falls below this value the current branch of
the TreeSearch procedure cannot go deeper.}
    {Process inputs and outputs in chronological order.}
    repeat
        Model.NextInputOutputEvent;
        if Model.IsOutputEvent then
            begin
                {Use output vectoring system to determine the better output.}
                SelectedOutput := OutputVectoringSystem.GetSelectedOutput (Model);
                {Do the output for real.}
            end
        end
    until false;
end.

```

```

        if not Model.IsPrioritizationControl then
            Model.MakeActualOutput (SelectedOutput);
            {Update the model.}
            Model.DoInputOutputEvent (SelectedOutput)
        end
    else
        {Get the input value and use it to update the model.}
        Model.DoInputOutputEvent (Model.GetActualInput)
    until false {i.e. just keep running}
end. {end of AISystem2 program}

```

Modelling System

```

Unit Modelling;
{Illustrates the Planning as Modelling concept for AI.}
{A modelling system which observes real or hypothetical inputs and outputs by
an AI system and provides probabilistic predictions of future inputs and
outputs.}
{Object Pascal (Delphi Programming Language)}
{Modelling.pas}
{December 2006}
{www.paul-almond.com}
{Copyright Paul Almond 2006. All Rights Reserved.}

interface

type

    TBit = 0..1;

    TModel = class

    private

        {Variable declarations specific to the modelling system go here.}
        {These are all internal to the modelling system - the only interaction
with code outside this unit is by using the methods defined here.}

    public

        constructor Create;
        {Creates an instance of the model and puts it in its initial state in which
no previous inputs and outputs have occurred.}

        procedure EnableUndo;
        {Directs the modelling system to store enough information to allow any
changes to it to be reversed.}
        {Allows changes to the modelling system by the output vectoring system's
tree search to be hypothetical.}

        procedure DisableUndo;
        {Directs the modelling system not to store enough information to allow
changes to be reversed and to discard any such information already stored.}
        {Used when the model is being affected by real, rather than hypothetical,
input and output events.}

```

```

function UndoEnabled: boolean;
{Returns true if undo is enabled, otherwise returns false. Included for
completeness and because it may be convenient for debugging in a real system.}

procedure NextInputOutputEvent;
{Makes the next input or output event (chronologically) after the current
input or output event the new current input or output event.}

function EventProbability (InputOutputValue:TBit): real;
{Returns the probability that a value of InputOutputValue (which is 0 or 1)
will be input or output when the current input or output event occurs.}

function IsOutputEvent: boolean;
{Returns true if the current input or output event is an output event,
otherwise returns false.}

function IsInputEvent: boolean;
{Returns true if the current input or output event is an input event,
otherwise returns false.}

function IsPrioritizationControl: boolean;
{Returns true if the current input or output event is a prioritization
control output event (also implying an output event), otherwise returns false.}

procedure UndoNextInputOutputEvent;
{Undoes the effects of procedure NextInputOutputEvent, i.e. makes the
previous input or output event (chronologically) before the current input or
output event the new current input or output event.}
{Requires undo to be enabled.}

procedure ApplyInputOutputEventToModel (InputOutputValue:TBit);
{Puts the model into the state that it should be in after the current input
or output event has occurred with an input or output of value InputOutputValue,
i.e. the relevant input or output bit is fixed.}
{This could be a real or hypothetical updating of the model.}

procedure UndoApplyInputOutputEventToModel;
{Undoes the effects of the last use of procedure
ApplyInputOutputEventToModel. Requires undo to be enabled.}

procedure ApplyPrioritizationControlToModel (OutputValue:TBit);
{Adjusts prioritization in the model, using the current input or output
event as a prioritization control output with value OutputValue.}
{Requires the current input output or output event to be a prioritization
control output.}

procedure UndoApplyPrioritizationControlToModel;
{Undoes the effects of the last use of procedure
ApplyPrioritizationControlToModel.}
{Requires undo to be enabled.}

procedure DoInputOutputEvent (InputOutputValue:TBit);
{Fixes the current input or output event as having occurred, actually or
hypothetically, with value InputOutputValue.}
{Updates the model accordingly so that any subsequent probabilities of
input or output events obtained from the model assume that the current input or
output event has occurred with value InputOutputValue.}

```

{If the input or output event is a prioritization control output then it applies prioritization to the model.}
 {Included for convenience. The effects of this procedure can be produced by using procedures ApplyPrioritizationControlToModel and ApplyInputOutputEventToModel.}

```
procedure UndoDoInputOutputEvent;
{Undoes the effects of the last use of procedure DoInputOutputEvent.}
{Requires undo to be enabled.}
{Included for convenience. The effects of this procedure can be produced by using procedures UndoApplyPrioritizationControlToModel and UndoApplyInputOutputEventToModel.}
```

```
function SituationalEvaluationFunction: real;
{Applies the situational evaluation function to the model and returns a score indicating the desirability of the situation described by the model.}
```

{INPUT / OUTPUT SUBROUTINES}

```
procedure MakeActualOutput (OutputValue:TBit);
{Sends an actual output to the outside world. Has no effect on the model itself.}
```

```
function GetActualInput: TBit;
{Receives an actual input from the outside world. Has no effect on the model itself.}
```

end; {end of TModel class}

implementation

{MODELLING SYSTEM SUBROUTINES}

```
constructor TModel.Create;
{Creates an instance of the model and puts it in its initial state in which no previous inputs and outputs have occurred.}
{Any probability values associated with future input or output events in the model will be determined accordingly - probably 0.5.}
begin
  {The code here is specific to the modelling system.}
end; {end of TModel.Create constructor}
```

```
procedure TModel.EnableUndo;
{Directs the modelling system to store enough information to allow any changes to it to be reversed.}
{Allows changes to the modelling system by the output vectoring system's tree search to be hypothetical.}
begin
  {The code here is specific to the modelling system.}
end; {end of TModel.EnableUndo procedure}
```

```
procedure TModel.DisableUndo;
{Directs the modelling system not to store enough information to allow changes to be reversed and to discard any such information already stored.}
{Used when the model is being affected by real, rather than hypothetical, input and output events.}
begin
```

```

    {The code here is specific to the modelling system.}
end; {end of TModel.DisableUndo procedure}

function TModel.UndoEnabled: boolean;
{Returns true if undo is enabled, otherwise returns false. Included for
completeness and because it may be convenient for debugging in a real system.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.UndoEnabled function}

procedure TModel.NextInputOutputEvent;
{Makes the next input or output event (chronologically) after the current input
or output event the new current input or output event.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.NextInputOutputEvent procedure}

function TModel.EventProbability (InputOutputValue:TBit): real;
{Returns the probability that a value of InputOutputValue (which is 0 or 1)
will be input or output when the current input or output event occurs.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.EventProbability function}

function TModel.IsOutputEvent: boolean;
{Returns true if the current input or output event is an output event,
otherwise returns false.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.IsOutputEvent function}

function TModel.IsInputEvent: boolean;
{Returns true if the current input or output event is an input event, otherwise
returns false.}
{Always returns the opposite of what IsOutputEvent would return.}
{Not used in this program and only included for completeness.}
begin
    IsInputEvent:= not Self.IsOutputEvent;
end; {end of TModel.IsInputEvent function}

function TModel.IsPrioritizationControl: boolean;
{Returns true if the current input or output event is a prioritization control
output event (also implying an output event), otherwise returns false.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.IsPrioritizationControl function}

procedure TModel.UndoNextInputOutputEvent;
{Undoes the effects of procedure NextInputOutputEvent, i.e. makes the previous
input or output event (chronologically) before the current input or output
event the new current input or output event.}
{Requires undo to be enabled.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.UndoNextInputOutputEvent procedure}

procedure TModel.ApplyInputOutputEventToModel (InputOutputValue:TBit);

```

```

{Puts the model into the state that it should be in after the current input or
output event has occurred with an input or output of value InputOutputValue,
i.e. the relevant input or output bit is fixed.}
{This could be a real or hypothetical updating of the model.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.ApplyInputOutputEventToModel procedure}

procedure TModel.UndoApplyInputOutputEventToModel;
{Undoes the effects of the last use of procedure ApplyInputOutputEventToModel.
Requires undo to be enabled.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.UndoApplyInputOutputEventToModel procedure}

procedure TModel.ApplyPrioritizationControlToModel (OutputValue:TBit);
{Adjusts prioritization in the model, using the current input or output event
as a prioritization control output with value OutputValue.}
{Requires the current input output or output event to be a prioritization
control output.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.ApplyPrioritizationControlToModel procedure}

procedure TModel.UndoApplyPrioritizationControlToModel;
{Undoes the effects of the last use of procedure
ApplyPrioritizationControlToModel.}
{Requires undo to be enabled.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.UndoApplyPrioritizationControlToModel procedure}

procedure TModel.DoInputOutputEvent (inputOutputValue:TBit);
{Fixes the current input or output event as having occurred, actually or
hypothetically, with value InputOutputValue.}
{Updates the model accordingly so that any subsequent probabilities of input or
output events obtained from the model assume that the current input or output
event has occurred with value InputOutputValue.}
{If the input or output event is a prioritization control output then it
applies prioritization to the model.}
{Included for convenience. The effects of this procedure can be produced by
using procedures ApplyPrioritizationControlToModel and
ApplyInputOutputEventToModel.}
begin
    Self.ApplyInputOutputEventToModel (InputOutputValue);
    if Self.IsPrioritizationControl then
        Self.ApplyPrioritizationControlToModel (InputOutputValue)
end; {end of TModel.DoInputOutputEvent procedure}

procedure TModel.UndoDoInputOutputEvent;
{Undoes the effects of the last use of procedure DoInputOutputEvent.}
{Requires undo to be enabled.}
{Included for convenience. The effects of this procedure can be produced by
using procedures UndoApplyPrioritizationControlToModel and
UndoApplyInputOutputEventToModel.}
begin
    if Self.IsPrioritizationControl then

```

```

        Self.UndoApplyPrioritizationControlToModel;
        Self.UndoApplyInputOutputEventToModel
end; {end of TModel.UndoDoInputOutputEvent procedure}

function TModel.SituationalEvaluationFunction: real;
{Applies the situational evaluation function to the model and returns a score
indicating the desirability of the situation described by the model.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.SituationalEvaluationFunction function}

{INPUT / OUTPUT SUBROUTINES}

procedure TModel.MakeActualOutput (OutputValue:TBit);
{Sends an actual output to the outside world. Has no effect on the model
itself.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.MakeActualOutput procedure}

function TModel.GetActualInput: TBit;
{Receives an actual input from the outside world. Has no effect on the model
itself.}
begin
    {The code here is specific to the modelling system.}
end; {end of TModel.GetActualInput function}

end. {end of ModellingSystem unit}

```

Output Vectoring System

```

Unit OutputVectoring;
{Illustrates the Planning as Modelling concept for AI.}
{Performs a look-ahead tree search to determine the optimum output. The search
is constrained by the modelling system.}
{This is not a planning system: the main planning is provided by the modelling
system in the form of the constraint.}
{Object Pascal (Delphi Programming Language)}
{OutputVectoring.pas}
{December 2006}
{www.paul-almond.com}
{Copyright Paul Almond 2006. All Rights Reserved}

interface

uses Modelling;

type

    TOutputVectoringSystem = class

    private

        {Variable declarations specific to the output system go here.}
        {These are all internal to the output vectoring system - the only
interaction with code outside this unit is by using the methods defined here.}

```

```
    CutOffProbability: real; {If the running probability falls below this value
the current branch of the TreeSearch procedure cannot go deeper.}
```

```
    {Currently the cut-off probability is the only data stored by the output
vectoring system, but other data could be added later.}
```

```
    function GetLowerBranchScore(Model:TModel; InputOutputValue:TBit;
RunningProbability:real): real;
    {Follows a branch of the search tree to a lower node, returning the score
that gets backed up to it.}
```

```
    procedure GetScoreFromLowerOutputs(LowerScore0,LowerScore1:real; var
Score:real; var SelectedOutput:TBit);
    {Decide what the score should be for this node, based on the two scores
backed up from branches to lower nodes for outputs of 0 and 1.}
    {Also returns the output (0 or 1) associated with the selected score,
though this is only of interest in the top level of the tree search.}
```

```
    function GetScoreFromLowerInputs
(LowerScore0,LowerScore1,Probability0,Probability1:real): real;
    {Decides what the score should be for this node, based on the two scores
backed up from lower nodes for inputs of 0 and 1.}
```

```
    procedure TreeSearch (Model:TModel; var SelectedOutput:TBit; var
Score:real; RunningProbability:real);
```

```
    public
```

```
    procedure SetCutOffProbability (NewCutOffProbability:real);
    {Sets the cut-off probability to NewCutOffProbability.}
    {If the running probability falls below this value the current branch of
the TreeSearch procedure cannot go deeper.}
```

```
    function GetCutOffProbability: real;
    {Returns the current cut-off probability which was set by procedure
SetCutOffProbability. Included for convenience and possible use in debugging.}
```

```
    function GetSelectedOutput (Model:TModel): TBit;
    {The main output vectoring system routine. Starts the recursive search to
determine the better output (0 or 1) for the current output event.}
```

```
end; {end of TOutputVectoringSystem class}
```

```
implementation
```

```
procedure TOutputVectoringSystem.SetCutOffProbability
(NewCutOffProbability:real);
{Sets the cut-off probability to NewCutOffProbability.}
{If the running probability falls below this value the current branch of the
TreeSearch procedure cannot go deeper.}
begin
```

```
    Self.CutOffProbability:= NewCutOffProbability;
end; {end of TOutputVectoringSystem.SetCutOffProbability procedure}
```

```
function TOutputVectoringSystem.GetCutOffProbability: real;
```

```

{Returns the current cut-off probability which was set by procedure
SetCutOffProbability. Included for convenience and possible use in debugging.}
begin
    GetCutOffProbability:= Self.CutOffProbability;
end; {end of TOutputVectoringSystem.GetCutOffProbability function}

function TOutputVectoringSystem.GetLowerBranchScore (Model:TModel;
InputOutputValue:TBit; RunningProbability:real): real;
{Follows a branch of the search tree to a lower node, returning the score that
gets backed up to it.}
var
    NullOutput: TBit; {The selected output is of no interest at this level of
recursion.}
    Score: real; {the score backed up from lower levels of the search tree}
begin
    Model.DoInputOutputEvent (InputOutputValue);
    Model.NextInputOutputEvent;
    Self.TreeSearch (Model, NullOutput, Score, RunningProbability);
    Model.UndoNextInputOutputEvent;
    Model.UndoDoInputOutputEvent;
    GetLowerBranchScore:= Score
end; {end of TOutputVectoringSystem.GetLowerBranchScore function}

procedure TOutputVectoringSystem.GetScoreFromLowerOutputs
(LowerScore0,LowerScore1:real; var Score:real; var SelectedOutput:TBit);
{Decide what the score should be for this node, based on the two scores backed
up from branches to lower nodes for outputs of 0 and 1.}
{Also returns the output (0 or 1) associated with the selected score, though
this is only of interest in the top level of the tree search.}
begin
    {Select the higher score and the relevant output.}
    if LowerScore0 > LowerScore1 then
    begin
        Score:= LowerScore0;
        SelectedOutput:= 0
    end
    else
    begin
        Score:= LowerScore1;
        SelectedOutput:= 1
    end
end; {end of TOutputVectoringSystem.GetScoreFromLowerOutputs procedure}

function TOutputVectoringSystem.GetScoreFromLowerInputs
(LowerScore0,LowerScore1,Probability0,Probability1:real): real;
{Decides what the score should be for this node, based on the two scores backed
up from lower nodes for inputs of 0 and 1.}
begin
    {Determine the mean (expected) score}
    GetScoreFromLowerInputs:= (LowerScore0*Probability0 +
LowerScore1*Probability1) / 2;
end; {end of TOutputVectoringSystem.GetScoreFromLowerInputs function}

procedure TOutputVectoringSystem.TreeSearch (Model:TModel; var
SelectedOutput:TBit; var Score:real; RunningProbability:real);
{Searches recursively to determine the better output (0 or 1) for the current
output event.}

```

```

var
  LowerScore0, LowerScore1: real; {Scores backed up from the level below.}
begin
  {Has the cut-off limit been reached yet?}
  if RunningProbability < Self.CutOffProbability then
  begin
    {Branch off for the 0 and 1 cases.}
    {Branch off for the 0 case, decreasing RunningProbability.}
    LowerScore0:= Self.GetLowerBranchScore (Model, 0,
RunningProbability*Model.EventProbability(0));
    {Branch off for the 1 case, decreasing RunningProbability.}
    LowerScore1:= Self.GetLowerBranchScore (Model, 1,
RunningProbability*Model.EventProbability(1));
    {Decide how to back up the score.}
    if Model.IsOutputEvent then
      Self.GetScoreFromLowerOutputs (LowerScore0, LowerScore1, Score,
SelectedOutput)
    else
      Score:= Self.GetScoreFromLowerInputs (LowerScore0, LowerScore1,
Model.EventProbability(0), Model.EventProbability(1))
    end
  else
    {This is a terminal node. Return a score for it.}
    Score:= Model.SituationalEvaluationFunction
  end; {end of TOutputVectoringSystem.TreeSearch procedure}

function TOutputVectoringSystem.GetSelectedOutput (Model:TModel): TBit;
{The main output vectoring system routine. Starts the recursive search to
determine the better output (0 or 1) for the current output event.}
var
  SelectedOutput: TBit;
  NullScore: real; {the score associated with use of this output - not used
at this level of the program but included because the output vectoring system's
routine requires it as a variable parameter}
begin
  Model.EnableUndo;
  Self.TreeSearch (Model, SelectedOutput, NullScore, 1);
  {1 is the running probability which will decrease with successive levels of
recursion depending on how likely the input or output branches are, until the
CutOffProbability threshold is reached.}
  Model.DisableUndo;
  GetSelectedOutput:= SelectedOutput
end; {end of TOutputVectoringSystem.GetSelectedOutput procedure}

end. {end of OutputVectoringSystem unit}

```

Appendix 4: An Example of Planning As Modelling in Chess

This appendix will show how planning as modelling could be used in chess. The method uses a modelling system with an output vectoring system that performs a tree search like that in a conventional chess algorithm, except modified to be constrained by the modelling system.

In chess a move of a piece by a single player is called a “half move” and is what most people would call a “move”. Using the term “half move” would confuse many readers: I will be calling it a “move”.

Storing the State of the Chess Board

The arrangement of pieces on a chess board, with some minimal extra information such as which player is to move next and whether players have castled completely characterises a situation in chess and we do not need to try to evaluate the model. In addition to its abstracted information used for predictions, the modelling system maintains a data structure storing information about the state of the chess game – the positions of the pieces on the board, the player to move next, etc – as with a conventional chess algorithm, and the situational evaluation function uses this to generate scores. Evaluation functions for chess situations are already widely available.

Input Events

Each input event corresponds to the making of a chess move by the AI system’s opponent. The possible values for an input are the possible chess moves that can be made by the system’s opponent.

Output Events

Each output event corresponds to the making of a chess move by the AI system. The possible values for an output are the possible chess moves that can be made by the AI system.

Encoding of Inputs and Outputs

Inputs and outputs are encoded in the same way, using any system for encoding chess moves. An obvious encoding is the algebraic chess notation in which a letter and digit give the starting square of the chess move and another letter and digit give the destination square, for example: “E2D4”, meaning move from square E2 to square D4.

The Modelling System

The modelling system is informed about input events or output events – chess moves – as they occur. It can also be hypothetically informed about such input or output events, making it simulate a possible future. The *current event* is the chess move happening “now” about which the system is being informed (possibly in simulating a hypothetical future) or about which the system is giving a probabilistic prediction. The modelling system has services to do the following:

- Inform the modelling system that the current event (current move) has occurred with a particular move being made.
- Obtain a probability value from the modelling system that a particular move (e.g. E2D4) will be made as the current event by the AI system’s opponent (for an input event) or the AI system itself (for an output event).
- Move the current event to the next move.
- Move the current event back to the previous move.

When the AI system’s opponent actually makes a move, this is a real input event. The modelling system is informed of the move made and changes its predictions accordingly, also updating the comparatively simple data structure storing the state of the chess game. The current event is then moved to the next move.

When the AI system is required to make a move, the output vectoring system searches for the best move, the search being constrained by the modelling system. When found, the best move is actually made and the modelling system informed of the move (so that it updates its predictions and the simple data structure storing the state of the chess game). The current event is then moved to the next move.

Output Vectoring System

The output vectoring system is used when the current event is an output event – that is to say, when the AI system must make a move.

The output vectoring system performs a look-ahead tree search with each node corresponding to an input or output event – that is to say, a chess move – starting from the current event – the current move. Each branch corresponds to a different possible move by the AI system or its opponent. This is similar to the tree search in a conventional chess program, the difference being that the search is constrained by the modelling system.

During the tree search the data structure storing the state of the chess game is updated as in a conventional chess algorithm.

Constraint of the Output Vectoring System

The *running probability* is set to 1 at the start of the tree search.

Each time the search goes down another branch, corresponding to a specific chess move by the AI system or its opponent, the output vectoring system requests the probability for that particular chess move occurring as the current event from the modelling system. The running probability is then multiplied by this value to give the new running probability for the next node. The modelling system is then hypothetically informed that the particular move corresponding to this branch has just been made for the current event. The current event is then moved to the next input or output event – the next chess move by the AI system or its opponent.

The search only continues while the running probability is less than the *cut-off probability*. If, at a node, this condition is not met then this node becomes a terminal node. This is the constraint from the modelling system working.

Terminal Nodes

At terminal nodes a situational evaluation function determines a score for the chess situation as in conventional chess programs [8]. The evaluation function is applied to the data structure storing the state of the board, as in conventional chess programs, rather than the model, as would usually be the case in planning as modelling. This score is then “backed-up” the search tree to previous nodes in the same way as in chess programs. That is to say, the score is made available at the node corresponding to the chess move immediately before the chess move for the terminal node.

Backing-Up of Scores

Backing-up of scores is the same as in conventional chess algorithms. For each node to which scores are backed-up from lower nodes a single score is assigned to that node which is in turn backed-up to higher nodes. In this way scores are backed-up from terminal nodes, through all nodes, to the top of the search tree. Scores are assigned to nodes from backed-up scores as follows:

- For an input event node (a node corresponding to a move by the opponent) the *lowest* score of all the backed-up scores is assigned to the node.
- For an output event node (a node corresponding to a move by the AI system) the *highest* score of all the backed-up scores is assigned to the node.

This way in which input event nodes are dealt with is an example of pessimism in the backing-up of scores rather than averaging: it is assumed that the opponent can make things as bad as possible. An argument could be made that having probabilities of moves in the modelling system might allow an alternative approach, but I will stay with the conventional approach here.

Providing the Best Move

The purpose of the output vectoring system’s search is to find the best chess move to use for the current event (before the output vectoring system’s search started). This is the chess move corresponding to the best score at the first node in the search tree, which the output vectoring system must provide with the score assigned to that node, at least for the first node.

Undoing Changes

When traversing a search tree the system will start at higher nodes and extend branches down to lower nodes. After completion of processing at the nodes below a particular node, processing will return to the higher node, to be extended down to different lower nodes later. This means that changes made at lower nodes need to be undone when the search returns to higher nodes.

A similar issue arises in conventional chess programs. A conventional chess program makes changes to the state of the chess board at lower nodes which must be undone as processing returns to higher nodes. One way of doing this would be to pass *copies* of data structures to lower nodes, rather than the data itself. Each time the tree search went down a branch to a node, all the data describing the chess board and the state of the game would be copied.

With planning as modelling this issue affects the same data structure describing the state of the chess game, but it also affects the entire modelling system, the running probability and the current event. Rather than suggest that all these should be passed as copies, I will use the approach taken when discussing services that the modelling system would generally need in planning as modelling (see Appendix 2) and assume that the modelling system provides services for this purpose. When processing in the tree search returns to a higher node from a lower node, services in the modelling system would be used to undo the affects on the model of informing it that the chess move corresponding to the lower node had occurred and to undo the advancement of the current event to the next move that had occurred when going to the lower node.

Appendix 5: Examples Relating to the Horizon Problem

Appendix 5.1: An Example of the Horizon Problem

Suppose that some node in the output vectoring system's search tree corresponds to a future output event with two possible output values: "0" and "1".

The output vectoring system obtains the relevant probabilistic predictions from the modelling system, which gives probabilities of 99% that the output value will be "1" and 1% that it will be "0". The modelling system considering a "1" output value more likely at this node means that an output value of "1" is more likely to be found preferable in this hypothetical situation.

The search will branch off from this node with separate paths for output values of "0" and "1" and will branch out from these new nodes. Two scores will be backed-up to this node from the "0" and "1" paths. The output vectoring system will decide which score is preferable – in the case of an output node at least. Let us consider two possible situations for the returned scores:

In the first situation a score of 27 is returned for the "1" output and a score of 15 for the "0" output. The search following on for the "0" output was shallower so could be considered less reliable, but it returned a low score anyway, so it is not worth spending more time on it. The expected result obtained for the "1" output appears better and was considered deeply in comparison with a shallow consideration of the consequences of the "0" output. We could be wrong: with a deeper search the "0" output might turn out to be better, but it had all the chance we were prepared to give it and it was not worth spending more processing resources on it. This may seem risky, but is no different from any other differential allocation of resources to research. A human manager, for example, will devote more resources to considering suggestions that seem better.

Let us now consider a second situation in which the search for the "1" output returns a score of 15 and the search for the "0" output returns a score of 27. This is a different situation. We have invested more computation in checking out the future consequences of making the "1" output than in checking out the future consequences of making the "0" output. We thought it was 99% likely that the "1" output would turn out to be better, but the "0" output has turned out to be better. We have expended less computational resources on checking out the "0" output: we used a shallower search due to the initially low value of the running probability. We are therefore going to choose to make the "0" output because it looks better, having only shallowly considered it. On deep consideration we may find that the score for the "0" output is less than the score for the "1" output. This is not the same as the previous situation because we expected the "1" score to be better anyway.

Appendix 5.2: Examples of Solving the Horizon Problem for General Systems

Example 1:

At an output event node we have these possible output values with the probabilities from the modelling system:

Output Value	0	1	2	3
Probability	0.15	0.16	0.05	0.2

Probability_{Max} is 0.6.

We will store the “Deep Search” flag for each possible output value, indicating whether or not the score that we have for it was obtained using a deep search. A deep search is one in which the search was extended from the current node, for that output value, with an effective probability of Probability_{Max}. We extend the search as in the previously described planning as modelling process. The probability given by the modelling system is used as the effective probability in each case, and the only output value for which the effective value is Probability_{Max} is “1”, the most likely output. If a cut-off probability is used then in each case the running probability is multiplied by the effective probability to give the running probability at the next node. For example, if the running probability is 0.07 then, with a cut-off probability in use, the running probabilities are as follows:

Output Value	0	1	2	3
Probability	0.15	0.16	0.05	0.2
Running Probability	0.07x0.15 =0.0105	0.07x0.6 =0.042	0.07x0.05 =0.0035	0.07x0.2 =0.014

Suppose the scores backed-up to this node from the lower branches of the search tree give the following situation:

Output Value	0	1	2	3
Probability	0.15	0.16	0.05	0.2
Running Probability	0.07x0.15 =0.0105	0.07x0.6 =0.042	0.07x0.05 =0.0035	0.07x0.2 =0.014
Deep Search	N	Y	N	N
Score	15	27	13	20

The highest scoring output value is “1”, with a score of 27, for which the “Deep Search” flag is set: the search for it was extended from this node with an effective probability of Probability_{Max}, which is 0.6. Therefore, this score should be accepted. In this situation it is assumed that an output of “1” would be selected at this node and a score of 27 is assigned to this node.

Example 2:

Let us imagine a situation like the previous one, except with different scores backed-up to the current node:

Output Value	0	1	2	3
Probability	0.15	0.16	0.05	0.2
Running Probability	0.07x0.15 =0.0105	0.07x0.6 =0.042	0.07x0.05 =0.0035	0.07x0.2 =0.014
Deep Search	N	Y	N	N
Score	24	20	10	27

The highest scoring output value is now “3” with a score of 27. As the “Deep Search” flag indicates, this is no longer the most likely output and was obtained with an effective probability less than $Probability_{Max}$ – actually a value of 0.2. We therefore extend the search from this node again, down the path for the “3” output value, with an effective probability of $Probability_{Max}$, which is 0.6, setting the “Deep Search” flag. If a cut-off probability is used then the running probability for this output changes from 0.014 to $0.07 \times 0.6 = 0.042$. Suppose a score of 23 is backed-up from this search, so the situation is now like this:

Output Value	0	1	2	3
Probability	0.15	0.16	0.05	0.2
Running Probability	0.07x0.15 =0.0105	0.07x0.6 =0.042	0.07x0.05 =0.0035	0.07x0.6 =0.042
Deep Search	N	Y	N	Y
Score	24	20	10	23

The deeper search for the “3” output value decreased the score and it is no longer the highest score. The highest score is 24, for the “0” output value. The “Deep Search” flag is not set for this score, however: it was only obtained with an effective probability of 0.15 – less than the $Probability_{Max}$ value of 0.6 – so the search needs to be deepened with an effective probability of $Probability_{Max}$ (0.6) and the “Deep Search” flag is then set for this output value. If a cut-off probability is used then the running probability for this output value now becomes $0.07 \times 0.6 = 0.042$. Suppose that when this is done a score of 19 is returned so the situation is now as follows:

Output Value	0	1	2	3
Probability	0.15	0.16	0.05	0.2
Running Probability	0.07x0.6 =0.042	0.07x0.6 =0.042	0.07x0.05 =0.0035	0.07x0.6 =0.042
Deep Search	Y	Y	N	Y
Score	19	20	10	23

The deeper search for the “0” output reduced its score, making it no longer the highest score. The highest score is now 23, for the “3” output, for which the “Deep Search” flag is set – this score was obtained with an effective probability of 0.6, which is $Probability_{Max}$. We therefore accept it. We

assume that in the situation corresponding to this node the system would make an output of “3” and we assign a score of 23 to this node.

This example shows a lot of searching and some readers may wonder what the point of planning as modelling is if we have to do all this anyway. This situation is contrived, however, to show how the horizon issue could be resolved. Situations like this will be infrequent. When they occur, it does not mean an entire tree search must be performed: the decision to deepen the search is only taken at the current node. For any search deepened in this way the decision about whether or not to deepen the search must be taken at every subsequent node and most often it will not be deepened.

Appendix 5.3: Examples of Solving the Horizon Problem for Binary Systems

Example 1:

At an output event node in the search tree these are the probabilities from the modelling system:

Output Value	0	1
Probability	0.2	0.8

For each output value we extend the search from the current node, using the probability for that output value as the effective probability, meaning the “1” output has a deeper search.

Suppose the scores backed-up from these searches are as follows:

Output Value	0	1
Probability	0.2	0.8
Score	15	41

The higher score is for the “1” output and was obtained using a search with the higher effective probability of 0.8, so this score is accepted. It is assumed that an output of “1” would be made in the situation corresponding to this node and a score of 41 is assigned to this node to be backed-up to other nodes.

Example 2:

We have a system like the one in the previous example, with the same probability values, but when the search is extended down each path the scores backed-up to this node give the following situation:

Output Value	0	1
Probability	0.2	0.8
Score	25	22

The higher score is 25, for the “0” output, but we do not trust it because it was obtained with the lower probability as the effective probability. We therefore extend the search from this node down the “0” path again, this time with an effective probability of 0.8. Suppose that a score of 24 is

backed-up to this node from this revised search. This is still greater than the other score of 22, so we would assume that an output of “0” would be made in this situation and assign a score of 24 to this node. Suppose instead, now, that a score of 21 is backed-up to this node from the revised search down the “0” path. This is less than the other score of 22 so we would now reject the “0” output, assuming instead that the output made in this situation would be “1” and assigning a score of 22 to this node.

Appendix 6: Previous Articles

Appendix 6.1: Previous Articles

Planning as modelling has been discussed in a number of previous articles during which it has been modified. These articles are as follows:

Occam's Razor Part 9: Representation and Planning of Actions in Artificial Intelligence, 29 July 2006

(<http://www.paul-almond.com/OccamsRazorPart09.pdf>) [4]

At this stage the approach was not called “planning as modelling”. The general idea was proposed, however, as part of a more general AI system, developed in earlier articles [13,14,15,16], based on a hierarchy of “meaning extraction algorithms”. Inputs and outputs would occur at the bottom level of this hierarchy. Meanings would be abstracted at higher levels and would affect probabilistic predictions at lower levels. The output vectoring system would interface with the bottom level of the hierarchy.

How AI Would Work, 4 September 2006

(<http://www.paul-almond.com/HowAIWouldWork.pdf>) [3]

The same idea as in *Occam's Razor Part 9: Representation and Planning of Actions in Artificial Intelligence* [4], with details of the modelling system from earlier articles included in the same article. The article attempted an overview of an AI system. The article featured prioritization control outputs.

AI as a Boundary System, 17 September 2006

(<http://www.paul-almond.com/AIAsABoundarySystem.pdf>) [5]

Discussed the *boundary system* view of the relationship between the output vectoring system, the modelling system and the outside world. No technology was proposed. Instead it was shown how in one view the output vectoring system can be regarded as the AI system itself and the modelling system can be regarded as just another part of the outside world. Prioritization control outputs, in this view, are just normal outputs which the system uses to operate the modelling system as a tool in the outside world. The method by which the system learns to prioritize its modelling is therefore the same method used for general learning.

Planning as Modelling in AI, 26 November 2006

(<http://www.paul-almond.com/PlanningAsModellingInAI.pdf>) [1]

Described planning as modelling independently of a particular model architecture, making it compatible with any modelling system meeting various requirements to allow it to interface with the output vectoring system. The interface between the output vectoring system and the modelling system is by means of a shared data structure of probability values corresponding to past and future input and output events from and to which the modelling system and output vectoring system can read and write information about past input and output events and predictions of future input and

output events. This data structure is what remains of the bottom level of the hierarchical system from when the model system was specified as a hierarchy.

Programming of Planning As Modelling in AI, 28 December 2006

(<http://www.paul-almond.com/ProgrammingOfPlanningAsModellingInAI.pdf>) [2]

Provided example computer programs for planning as modelling in Standard Pascal and Object Pascal (The Delphi Programming Language). The interaction between the output vectoring system and modelling system was modified to rely on services or methods in the modelling system. That is to say, processes were defined for informing the modelling system about past input and output events and for the output vectoring system to obtain probabilistic predictions from the model. The idea of *the current event* was introduced, which takes advantage of the fact that only a single input or output event – the next one due to happen now in some real or hypothetical situation – needs to be considered at once.

Resolving the Horizon Problem in Planning As Modelling, 30 March 2007

(<http://www.paul-almond.com/ResolvingHorizonProblem.pdf>) [6]

Described the horizon problem and suggested methods of resolving it.

Appendix 6.2: How Planning As Modelling Has Evolved

A summary of how planning as modelling has developed over the above series of articles is as follows:

Description of the Basic Idea

The general idea of planning as modelling was discussed in *Occam's Razor Part 9: Representation and Planning of Actions in Artificial Intelligence* (<http://www.paul-almond.com/OccamsRazorPart09.pdf>) [4].

Inclusion of Inputs

In earlier articles about planning as modelling [3,4,5], the search performed by the output vectoring system was described as a search of the set of possible future sequences of outputs: inputs were omitted. *Planning as Modelling in AI* (<http://www.paul-almond.com/PlanningAsModellingInAI.pdf>) [1] corrected this, describing the output vectoring system as searching the set of possible future sequences of inputs and outputs.

Independence from Model Architecture

The first articles on planning as modelling [3,4,5] described it in the context of a hierarchical AI system. *Planning as Modelling in AI* (<http://www.paul-almond.com/PlanningAsModellingInAI.pdf>) [1] gave a more general view of planning as modelling, independent of the context of any particular modelling system. This does not mean that

the modelling system should not be hierarchical – merely that planning as modelling is not concerned with how the modelling system works.

Interaction Between Modelling System and Output Vectoring System Using Services

The interaction between the output vectoring system and the model was originally described in terms of the input and output history and prediction collection or combined probability collection. These can be considered as data structures containing sets of previous and future inputs and outputs into which we put observations of known inputs and outputs for the modelling system to observe and from which we extract its probabilistic predictions.

These data structures are a hangover from when planning as modelling was being described solely in the context of a hierarchical system. They are what is left of the bottom level of this hierarchy, but instead of saying that they must be the bottom level of a hierarchical system we say that they are used as the interface with a modelling system that works in a unspecified way.

Programming of Planning as Modelling in AI (<http://www.paul-almond.com/ProgrammingOfPlanningAsModellingInAI.pdf>) [2] introduced an improvement in the way that interaction with the modelling system is described. Rather than use data structures containing entire sets of probabilities we need only deal with a single input or output event at a time and the modelling system merely needs to provide services to this effect.

Dealing with the Horizon Problem

In *Resolving the Horizon Problem in Planning As Modelling* (<http://www.paul-almond.com/ResolvingHorizonProblem.pdf>) [6] the horizon problem and methods of resolving it were discussed.