Language for Description of Worlds

Dimiter Dobrev
Institute of Mathematics and Informatics
Bulgarian Academy of Sciences
d@dobrev.com

We will reduce the task of creating AI to the task of finding an appropriate language for description of the world. This will not be a programing language because programing languages describe only computable functions, while our language will describe a somewhat broader class of functions. Another specificity of this language will be that the description will consist of separate modules. This will enable us look for the description of the world automatically such that we discover it module after module. Our approach to the creation of this new language will be to start with a particular world and write the description of that particular world. The point is that the language which can describe this particular world will be appropriate for describing any world.

Keywords: Artificial Intelligence, Language for description of worlds, Event-Driven Model, Definition of Property, Definition of Algorithm.

Introduction

Our task is to understand the world. This means we have to describe it but before we can do so we need to develop a specific language for description of worlds.

The starting point of our language development process is answering the question "What actually is the world?" It will be shown that the world can be thought of as a function f, which is a function from $\mathbb{R}$ to $\mathbb{R}$. The f function and the initial state of the world are sufficient to provide a full description of the world. Such a description would be deterministic, which is good news because our aim is to predict the future as accurately as possible. The bad news is that this description may prove overly complicated and difficult to find. Moreover, a complicated description is not very credible because Occam's razor requires us to select the description which is the simplest.

In order to find a more simple description, we will admit a degree of randomness in the world and will look for a non-deterministic description. To make the f function non-deterministic we will substitute the initial state of the world with a set of states.

The set of states is a rough description of what we know about the state of the world. A more detailed description will be obtained when we have multiple sets of states and assign to each set the probability of the world's state being within that very set of states. In our terminology, the partitioning of the set of states in disjoint subsets will be a *question*, while the resulting subsets will be *basic answers*. The *answer* to a *question* will be the probabilities we have assigned to the *basic answers.*

We will define the non-deterministic function f through an *answer* to a *question*. Then we will sophisticate the definition by presenting it through an answer to a *group* of *questions* (in this article the term *group* will be synonymous to *set*).

We will assume that we do not ask irrelevant questions. For a question to be relevant, the answer to that question should be i) dependent on the past and ii) have some impact on the future. The definition of the world will depend significantly on the group of questions we have asked. We will assume that we have asked sufficiently many questions so that we can present the world as a Markov Decision Process (MDP) which satisfies the Markov property.

In this article we present the world (the f function) in a quite complicated way, using some convoluted and unintelligible formulae. The only purpose of these formulae is to give us an idea of how the world we trying to describe looks like. These formulas will not be involved in the creation of AI and will not be employed by AI itself in its reasoning.

We will not try to find the f function as such but an approximation of that function, which will be referred to as function g. The approximated function also will depend on the group of questions, however, here the questions will be abstract. In other words, we will not be interested in how we partition the set of states of the world. Instead, all that will be important is how many questions are out there and how many are the answers to each question. That is, how many times we have partitioned the set S to basic answers and how many basic answers are there in each of these partitions.

So the g function describes the world. But is the g function computable? If the function were computable, the language which describes the world would be some form of programming language. We will find out that the computable functions are not enough and therefor we will describe a somewhat broader class of functions.

We will show that once we have the language for description of worlds, creating Artificial Intelligence becomes easy. In order to understand the world, we need to find its description. One way to do this is by an exhaustive search (brute-force search) in all possible descriptions. This approach is foredoomed to be unsuccessful. We will take a smarter approach by assuming that the description of the world consists of individual modules which can be discovered one by one.

How many agents will live in the world? Each world has at least one agent – the protagonist. We will consider two versions of the world we are going to describe. In the first version, the protagonist will be alone and will play against himself. In the second scenario there will be another agent who will play against the first one.

Although the new language will not be a programming language, we will create it using the same approach that we would employ for developing a new programming language. We will start with a particular problem (i.e. a specific world) and will try to write a program which solves this problem (provide a description of the specific world).

The specific world we are going to describe will be the one in which the agent plays chess. We have written a program [8] which contains a simple description of this world and by this description emulates the world. Run this program and see how simple that description (of the chess game) happens to be. The description consists of 24 modules presented in the form of Event-Driven (ED) models. In addition to the ED models there are also two moving traces (i.e. two arrays). The ED models of the chess game belong to three types: (5 patterns + 9 algorithms + 10 properties = 24 models). We added also seven simple rules which we need as well.

It is important that our agent will not be able to see everything in this world. Rather than seeing the whole chessboard, the agent will only be able to see one square at a time. Thus we will be exploring a Partial Observability world, which is more interesting because if the agent were able to see everything the task would not be so stimulating. When the agent can only see a fragment of the picture, he will need to figure out how the rest of the picture (the one he cannot see) looks like.

We will use simple coding in order to limit the number of the commands given by the agent. To this end, we will divide the steps in three groups (the step number modulo 3). The specific command will depend on the step at which the agent gives that command.

This type of coding will provide the first pattern, namely "1, 2, 3". Similar to all other patterns in the world, the "1, 2, 3" pattern will be presented through an Event-Driven model. An ED model is an oriented graph consisting of several states and arrows between the states. The arrows are associated with events which in turn are presented as conjunctions. An event can be *true* sometimes and *false* at other times.

We will assume that something special occurs in certain states of the ED models. In our terminology this special occurrence will be a *trace*. The trace is what makes the model meaningful because if all states were the same it would not matter which state we are currently in.

The next two patterns will be Horizontal and Vertical. These patterns will provide the $\boldsymbol{x}$ and $\boldsymbol{y}$ coordinates of the observed chess square. The Cartesian product of these two models will be the model of the whole chessboard.

The trace of this Cartesian product will be the chessboard position. It will be a "moving" trace because the chessboard position is not still but keeps evolving.

Now that we have described the chessboard we should describe how the chess pieces move. This is where we need algorithms. What is an algorithm? We presented the patterns as Event-Driven models and will likewise use ED models to present algorithms. In our understanding, an algorithm is a sequences of actions (events). The programs which calculate $\mathbb{N} \rightarrow \mathbb{N}$ functions will be algorithms as per our definition. The definition will, however, be broader because it will include other algorithms such as cooking recipes.

We will describe the algorithms which determine how the chess pieces move. For example, the king's algorithm is quite simple because it is deterministic, while the rook's algorithm is more complex because it contains non-deterministic furcations.

Very few authors have ventured to define the term *algorithm*. The only attempts at an algorithm definition known to us are those of Moschovakis [1, 2].

In order to demonstrate that our definition of an algorithm makes sense, we will present the Turing machine as an Event-driven model. We will thus show that our definition covers the meaning which other authors imply in the term *algorithm*.

We will present the Turing machine as a separate world. The description of that world will consist of two ED models, the first of which will describe an infinite tape and the second model will describe the Turing machine as such. A proper algorithm will be the second ED model because the first one simply describes a tape of infinite length. An algorithm does not make sense unless we have something to run the algorithm on. Likewise, a Turing machine will not run unless we supply an infinite tape for the machine to run on.

Another fundamental term introduced in this article is *property*. This concept will also be a pattern and will also be presented through an Event-Driven model. We will use properties in order to code the agent's input. Although the input will consist of as little as four characters, these four characters will enable us present infinitely many properties.

The first world which we described is deterministic and for our description we used a g function which is deterministic. Most worlds however are not deterministic, hence our language for describing worlds should also be able to describe non-deterministic functions. As soon as we introduce a second agent, the world becomes non-deterministic because the exact behavior of the second agent is unknown.

The description of the world will include various events. Most of these events will be linked to the input and to the output. Certain events will be linked to the states of certain ED models. There will also be random events which occur with a certain probability. The description will go as far as allowing for impossible events. These will be events which never occur in reality, but can occur in our imagination. Regardless of their impossibility, these events will help us describe the world because they will be intertwined in various algorithms.

The first world we described included just one agent (protagonist) playing against himself. We will describe a second version of the chess game where, in addition to the protagonist, there will be a second agent (antagonist). Importantly, the state of the world will be different for the two agents. The state of the world is the state in which each agent is. The context is: "who", "where", "when". While we can reasonably assume that the "when" context is the same for all agents, the "where" context may vary across agents. The "who" context may also vary as some agents may be male while others may be female. One agent can play with the white pieces and another agent may play with the black pieces.

We have written a program which emulates the first world. Can we write such a program for the second world as well? Generally speaking, the answer is "no" because in that word there are statements such as "this algorithm can be executed" (i.e. "termination is possible") and operations such "agent executes algorithm". In the general case, these are undecidable problems and non-computable operations. So we said that in the general case the world can be a non-computable function meaning that in certain cases one can describe the world but cannot write a program which emulates the description. The particular case of the chess game is different. In that particular case, one can create an emulating program because everything in the chess game is finite and everything is computable.

How does the world look like?

Let us first see how the world looks like in the eyes of the agent. The agent sees a data stream or a series of inputs and outputs (observations and actions). This is how the stream looks like:

$$v_0, a_1, v_1, a_2, v_2, a_3, v_3, \ldots$$

So the world looks like (appears to be) a data stream. The appearance however covers the underlying essence of the world which the agent should endeavor to understand.

What is the underlying essence? This is the function which generates the data stream. More precisely, it is a function which takes its arguments from actions (the outputs) and returns its values in the form of observations (the inputs):

$$v_i = f(a_i)$$

This description is oversimplified because we assumed that the *f* function does not have a memory and does not depend on what has happened before. We assumed that the *f* function depends only and exclusively on the last action of the agent. This excessive simplification forgoes the core of the problem. Nevertheless, we do mention this possible simplification because it is used in nearly all articles dedicated to AI. In literature, the presence of this simplification is termed Full Observability and, accordingly, its absence is denoted as Partial Observability. In this article we will assume that the *f* function has a memory and accordingly we will be dealing with Partial Observability.

This is how the *f* function with a memory looks like:

$$< s_{i+1}, v_i > = f(s_i, a_i)$$

Here s_i is the state of the world. The state of the world is the memory of the world. But, the *f* function is the world proper, meaning that s_i is the memory of the *f* function. In other words, s_i is what the world has memorized at moment i.

The *f* function takes the current state of the world and the agent's action as an argument, and will return two values (in form of a 2-tuple). The first value returned will be the new state of the world (the new value of the memory) and the second one will be the new observation.

Can we assert that the *f* function describes the world and is therefore its underlying essence? Not quite, because we also need s_0. Hence, we need to add the initial state of the world (or the initial state of the memory). Now we can say that the pair $<f, s_0>$ is the world and that at least it describes the world precisely.

Note: We will assume that the agent's action and observation are letters from a finite alphabet and the state of the memory is a real number. In other words we will assume that the amount of information received and transmitted at each step is limited while the world's memory is the continuum. We need a continuum size to ensure that the memory of the world can store an infinite sequence of 0's and 1's (the number of these sequences is the continuum).

We do not need sizes larger that a continuum because any *f* function the memory of which is larger than a continuum can be expressed as function with a continuum memory. Let us take the equivalence relation "undistinguishable states" (i.e. two states are equivalent if it does not matter which state we start from). The size of the factor set of this relation is not larger than the

continuum. We can express f as a function over that factor set instead of a function over S. In this way we will present f as a function with a continuum memory.

All continuum-sized sets are isomorphic among themselves. Hence, we can choose any one of these sets. And we selected the most prominent of these sets – the set of real numbers $\mathbb{R}$ and therefore we assume that $S = \mathbb{R}$.

Note: We will make the picture more colorful and interesting by assuming that not all actions of the agent are allowed. Therefore, certain actions will be allowed at certain moments and other actions will be allowed at other moments. For this purpose we will assume that the f function is not a total one and therefore it is undefined for some "state of the world – action" tuples. This is how from the partial function f we will obtain the total function f_1:

$$f_1(s,a) = \begin{cases} f(s,a), & <s,a> \in dom(f) \\ <s, undef>, & <s,a> \notin dom(f) \end{cases}$$

Here *undef* is a special constant equal to the agent's observation when he tries to play an incorrect move.

A simple non-deterministic world

Although we did a good job defining the world as a deterministic function, this is inconsistent with our idea that the world is non-deterministic and involves randomness. How can we introduce randomness? By replacing the concrete states of the world with sets of states. When we know exactly which state the world is in, the f function will be deterministic and we will be able to tell exactly the next state of the world and the next observation. It may also be the case that we know the state of the world only approximately rather than exactly. In this case we will have a set of states instead of a single state.

We will also have to introduce probability. We must be able to tell the probability of the various subsets of $\mathbb{R}$. For this purpose we will add some probability space over $\mathbb{R}$ and designate that space as Ω.

$$\Omega = <\mathbb{R}, F, P>$$

F here will be the set of the measurable subsets of $\mathbb{R}$ (those which have a probability value) and P will be a function which operates on F and for each measurable subset returns its probability.

But, we want to have probabilities for all subsets of $\mathbb{R}$, not only for the measurable ones. Let $M \subset \mathbb{R}$ be a non-measurable subset. We will now say that the probability of M is the interval $[a, b]$, where a is the supremum of the probabilities of all measurable sets which are subsets of M and b is the infimum of the probabilities of all measurable sets which are supersets of M. (In literature, a is referred to as "inner measure" and b is referred to as "outer measure".) So, for each set of states we have defined either an exact probability or an interval of probabilities.

In order to define the f_4 function – which is the non-deterministic version of the f function – we will need two more versions of f:

$$f_2(M, a, v)=\{ s \mid \exists s' \in M, < s, v > = f_1(s', a) \}$$

This is the image of M after we have performed action a and have seen observation v.

$$f_3(M, a, v_i) = <M', p_i> = <f_2(M, a, v_i), \alpha.P(M_i)>$$

The f_3 returns a set and a probability value. The set is the image of M and p_i is the probability that the next observation will be v_i.

We have $n+1$ possible results (n possible observations and one possibility that a is an incorrect move). Action a partitions set S into $n+1$ disjoint subsets A_i.

$$A_i = \{ s \mid < s', v_i > = f_1(s, a) \}$$

From these subsets we obtain $M_i=M \cap A_i$. Each M_i set has its own probability or interval of probabilities, namely the probability of the result v_i. Here v_i traverses all possible observations. The last possible result is *undef*. In this case the state of the world remains the same, i.e. the M set is reduced to M_{n+1}. Therefore, the next set of states will be $M_{n+1}= f_2(M_{n+1}, a, undef)= f_2(M, a, undef)$.

The probability of each result is the probability of the corresponding set M_i. However, these probabilities should be normalized, i.e. their sum should be 1. To do so, we will move to a probability which is relative to the probability of M by multiplying it by some normalizing constant α. In our case, normalization is the transition to the conditional probability under the condition M. This will be done by multiplying by some constant α. If the probability of M is p, then the conditional probability will be the obtained by dividing by p. If the probability of M is in the interval $[a, b]$, then we have to divide by b.

From the f_3 function we will easily obtain the non-deterministic function:

$$f_4(M, a)$$

This function returns $n+1$ possible results in the form of $< f_2(M, a, v_i), v_i>$ and each result will be returned with the probability p_i, which is determined by f_3.

We presented the deterministic world as the 2-tuple $<f, s_0>$, so the simple non-deterministic world will be the following 3-tuple:

$$< f_4, M_0, \Omega>$$

Here M_0 is some set of states of the world (the initial set) and Ω is some probability space. There are many probability spaces which can be chosen for Ω and the various spaces will produce various non-deterministic worlds.

What is a question?

We described the state of the world in an extremely simplified way – by partitioning the set S in two parts (M and $S \setminus M$) and saying that the probability the current state to be within the first set is 1 and to be within the second set is 0.

A better way of describing the state of the world would be to partition S into finitely many disjoint sets Q_i (or into a countable number of such disjoint sets) and match each Q_i set with a probability or an interval of probabilities.

Relevant definitions:

Definition: A *question* is the partitioning of set S into a finite or countable number of disjoint subsets Q_i, where $\forall i$ $P(Q_i)\neq 0$. Each Q_i subset will be a *basic answer*.

Definition: An *answer* to a *question* is some congruence which matches each *basic answer* with a probability or an interval of probabilities.

Definition: A *group of questions* is a finite or countable set of questions.

Definition: An *answer to a group of questions* is the set of the answers obtained from a *group of questions* wherein each question in the group of questions has exactly one answer.

Definition: *"Do not know"* is the *answer* to a *question* where each *basic answer* to this question is matched with the probability interval *[0, 1]*. The answer to a *group of questions* is *"Do not know"* if the answers to all questions in the group are *"Do not know"*.

Definition: An *answer* to a *question* will be deterministic if the probability of one of its basic answers is 1 and the probabilities of all other basic answers are 0. The answer to a *group of questions* will be deterministic if the answers to all questions in the group are deterministic.

Arriving at a definition on the back of a question

We will describe the state of the world by answering a certain question. It can be assumed that the answer AQ to question Q transforms the probability space Ω into a new probability space Ω' where the probability of each set Q_i (of each basic answer) is multiplied by some factor q_i. How shall we calculate these factors?

$$q_i = AQ(Q_i)/P(Q_i)$$

When the value of Q_i in AQ is a number and the probability of Q_i is also a number, then q_i is a number, too. If these are the intervals of probabilities *[a', b']* and *[a, b]*, then q_i is the interval of factors *[a'/b, b'/b]*.

We want get the new idea of the state of world in the form of an answer to a question. We cannot seek answers to all questions because they are uncountably many. Therefore, we will choose a new question N and try obtain the new idea of the state of world as an answer to the question N.

What is the probability p_i for the next observation to be v_i, if the action is a?

$$p_i = P'(A_i) = \sum_r q_r . P(Q_r \cap A_i)$$

That probability is equal to the probability of the set A_i in the probability space Ω', which can be expressed as the sum shown above.

Which new answer AN_i to the question N will describe the state of the world provided that we (i) have started from a state described by the answer AQ, (ii) have performed action a and (iii) have seen observation v_i?

$$AN_i(N_j) = \alpha_i . \sum_r q_r . P(f_2(Q_r, a, v_i) \cap N_j)$$

Here r traverses the basic answers to Q, j traverses the basic answers to N and the constant α_i normalizes the answer AN_i.

From the f function we will develop one more function and will name it f_5.

$$\langle AN_i, p_i \rangle = f_5(AQ, a, v_i, N)$$

Here AN_i is an answer to the question N and p_i is a probability or an interval of probabilities. The value of p_i is equal to the probability that the next observation is v_i. That value does not depend on question N. The answer AN_i is the one which will describe the new state of the world if the observation is v_i.

Arriving at a definition on the back of multiple questions

Thus far we described the state of the world using the answer to a single question. Now let's replace the single answer with a group of answers. Will examine the case where (i) the group consists of two questions and (ii) the answers do not contain probability intervals (i.e. each basic answer comes with an exact probability).

Let's have the two questions B and C and their basic answers B_r and C_s. Let's also have some factors k_{rs} which depend on the questions only and not on the answers.

$$k_{rs} = \frac{P(B_r \cap C_s)}{P(B_r) . P(C_s)}$$

Let

$$q_{rs} = \frac{P'(B_r \cap C_s)}{P(B_r \cap C_s)} = \frac{AB(B_r) . AC(C_s) . k_{rs} . t_{rs}}{P(B_r) . P(C_s) . k_{rs}}$$

Here P' is the new probability in the new probability space Ω' which we obtain after considering the answers AB and AC to the questions B and C. Now we have to find the factors t_{rs}. How can we find them?

When some of the factors k is 0, then the corresponding factor t is also 0. When all factors k are 1, then all factors t will be 1, too. In the general case, the factors t depend on the answers and we will calculate them using the following equalities:

$$P'(B_r) = AB(B_r) = \sum_{s} P'(B_r \cap C_s) \qquad (1)$$

$$P'(C_s) = AC(C_s) = \sum_{r} P'(B_r \cap C_s) \qquad (2)$$

We will easily find factors b_r which satisfy equalities (1). Then we will easily find factors c_s which satisfy equalities (2). Unfortunately, when we fix equalities (2) we will unfix equalities (1). We will assume that we have found factors b_r and c_s such that when $t_{rs}=b_r.c_s$ then both (1) and (2) are satisfied. Here we do not explain how we will find these factors.

Thus, we have found the factors t_{rs} and thereby the factors q_{rs}. So we can extend the f_5 function and now it will use an idea of the state of the world described through an answer to a group of questions (rather than to a single question as per the previous section).

$$< AN_i, p_i> = f_5(AG, a, v_i, N)$$

Here p_i and AN_i are:

$$p_i = P'(A_i) = \sum_{r\,s} q_{rs}.P(B_r \cap C_s \cap A_i)$$

$$AN_i(N_j) = \alpha_i.\sum_{r\,s} q_{rs}.P(f_2(B_r \cap C_s, a, v_i) \cap N_j)$$

A real non-deterministic world

It is very important what questions we ask. As we said, we cannot afford to ask all questions because they are uncountably many. Therefore, we will assume that (i) we have fixed the group of questions *GQ* and (ii) both the current state of the world and the new state of the world are described by an answer to that group of questions. We will define the function f_6 such that it depends on *GQ* although *GQ* is not an argument of f_6.

The f_5 function provides us with an answer to a single question and enables us find answers to all questions in the *GQ* group (one at a time). Thus, from f_5 we obtain f_6.

$$< AG_i, p_i> = f_6(AG, a, v_i)$$

Here *AG* is an answer to the questions in the *GQ* group. The answers AG_i are also answers to the *GQ* questions.

Using the deterministic function f_6 we will construct a non-deterministic function.

$$f_7(AG, a)$$

The f_7 function will (i) return $n+1$ possible answers in the form of $<AG_i, v_i>$ and (ii) each of these answers will be returned with the corresponding probability p_i.

So far, we presented the deterministic world as the 2-tuple $<f, s_0>$ and the simple non-deterministic world as the 3-tuple $< f_4, M_0, \Omega>$. Now, the real non-deterministic world will be the 4-tuple

$$< f_7, Answer_0, GQ, \Omega >$$

Here $Answer_0$ is some answer to the group of questions GQ (the initial answer we will start from). We will assume that the initial answer is possible. An example of an impossible answer is where the answer is deterministic and the corresponding factor k_{rs} equals 0. In other words, an impossible situation is where (i) questions B and C cannot have answers r and s at the same time and (ii) another possibility does not exist.

We could replace the group of questions with a single question which has many basic answers, but we will not do this because we prefer to have many questions each of which has only a few basis answers.

Irrelevant questions

The non-deterministic world hinges heavily on GQ (the group of questions we have chosen). If we chose an empty group of questions $\varnothing$, we will have only one possible answer: the empty answer $\varnothing$. In this case we can assume that we have only one state of the world and enjoy Full Observability.

The more questions we include in the group and the further we partition the S set, the more accurate our description of the f function will be. For example, if the partitioning coincides with the factor set of the relation "undistinguishable states" and if the initial answer is deterministic, then we will obtain a deterministic world.

How many questions should we ask? Are there irrelevant questions? Yes. For a question to be relevant, its answer must somehow depend on the past and somehow influence the future. If the answer to the question does not anyhow depend on the past or does not anyhow influence the future, then the question is irrelevant.

An example of an irrelevant question is "Which side of the coin will fall upside? Heads or tails?". Despite all the importance of that question, we will not include it in the group because the answer to the question does not anyhow depend on what has happened before. In other words, the answer to the question does not depend on the past and we cannot know the answer before we toss the coin.

Another example is "How many beans of peas are there in my dish? Odd or even number?". The answer to that question has no impact on the future whatever. Neither does the answer depend on the past, unless we have made the effort to count the pea beans.

Note: A question is irrelevant if it contains two needlessly separated basic answers. They may be needlessly separated from the perspective of the past in case that regardless of the previous answer, of the action performed and of the observation seen, the two basic answers obtain always the same factor q. They are needlessly separated from the perspective of the future in the

following case: If we take the two deterministic answers which provide 1 to those basic answers then these two deterministic answers produce one and the same forecast for the future. Therefore, these basic answers do not anyhow affect the future.

We will assume that we will not be asking any irrelevant questions. It follows therefore that the classes of the relation "indistinguishable questions" cannot be partitioned any further because the answer to the question which leads to such partitioning does not anyhow affect the future.

We presented the non-deterministic world as a Markov Decision Process (MDP). There are three minor differences from the classic MDP definition:
1. The set of states here is the set of intersections between basic answers. The latter set may not always be finite. It may be infinite or even reach the size of a continuum.
2. The probabilities are replaced with probability intervals.
3. We do not have rewards here because the objective is only to find a model while rewards are needed when we try to find a strategy.

Does our MDP satisfy the Markov property? This property means that the MDP model cannot be further improved. The property will be satisfied when we have asked a sufficient group of questions.

Definition: A group of questions is sufficient when every next question is either irrelevant or incapable to produce further partitioning of the set S.

The Markov property says that the future depends only on the state, while the way by which we have arrived at that state does not matter. Therefore, every next question which leads to further partitioning will either be independent from the past or will not affect the future.

We will assume that GQ is sufficient group of questions. Otherwise the description of the world would be rough, but this is not enough for us because we aim at a description which is maximally precise. For the same reason will also assume that the initial answer is deterministic.

Now that we have an idea about how the world we aim to describe looks like, we should find a way to make this description.

How shall we describe the world?

Thus far we have described the world in a very complicated way. We used the f function and a group of questions. The only purpose of that complicated description was to give us an idea about how the world we are trying the describe looks like. For arriving at the actual description of the world we will use a very straightforward g function.

First, the questions we are going to ask will be abstract ones, meaning that we will care only about the number of questions and the number of answers to each question. We will not care about the way the S set has been partitioned to basic answers or about the probability space Ω from which we would have obtained the probabilities.

All we know about the world (the f function) is our life in the world (the sequence of inputs and outputs). We will aim to find the g function on the basis of this information in the hope that g will successfully predict the future behavior of the world. Because life is finite, we can find a

computable deterministic function g which provides a precise description of life until now. Such a function may turn out to be excessively complicated which is not good because overly complicated functions are highly unlikely to produce a precise description of the future (Occam's razor principle). If we let randomness come in, then we may be able to find a much simpler non-deterministic function g which is still capable to describe life albeit with an approximation because the description thus obtained would include randomness. We will try to reconcile the two requirements and obtain a g function which is both maximally simple and maximally deterministic.

We will describe the state of the world through the answers to the questions asked. If the number of questions is countable, the answers will be a continuum. In this case only part of the answers will be describable and we will assume that the g function operates only with the describable answers. Thus, the g function is defined only for the describable answers and always returns a describable answer.

In order to describe the g function, it would be sufficient to describe the behavior of this function with deterministic answers. In order to extend the g function to non-deterministic answers, we will need the factors k_{rs}, but will typically assume that the answers are independent, i.e. the k_{rs} factors are equal to 1. In certain cases we may assume that a combination of answers is impossible or highly improbable, i.e. the corresponding factor is zero or close to zero.

While we will be aiming to find the right set of questions, we will never be sure about the sufficiency of the questions or about the absence of any irrelevant questions.

There are two reasons why we will not look for the f function and will only try to find its approximation, the g function.

1. The f function may not be describable, i.e. the function may not have a description which can be found.
2. The f function may be describable, but too complicated for us to find its description. That is why we will be looking for an approximation which is simpler although not very precise.

The general form of the g function is:

$$< Answer', p > = g(Answer, a, v)$$

where *Answer* is an answer to the questions asked, *Answer'* is the new answer obtained after we have played a and have observed v, and p is a probability or an interval of probabilities.

Computability of the world

The first question we are going to ask as we proceed to describe the world is whether the f function is computable. The answer is that the function is probably non-computable. The second question is whether the g function – which describes the f function – must be computable?

We can imagine the world and the agent as two black boxes which exchange information. The function which describes the agent must be computable, otherwise we would not be able to write a program which computes this function and will therefore fail to create AI. On the other hand, we can comfortably live with a non-computable g function (the function which describes the

world). We are not going to create the world because it is already there and all we need to do is describe it. However, can we describe a non-computable function? Yes, we can, but of course we would not be able to compute what we have described.

Indeed, in our own perception the world is non-computable. Let us take as an example the following rule: “A statement is true if there is proof for that statement”. This rule is non-computable because the question about the existence of proof is semi-decidable.

If we had to create the world (i.e. emulate it with a computer program), what would we do? We would need a *g* function which, besides being computable, is computable in reasonable time. For example, in a chess game each position is either winning or non-winning. While we can calculate whether a position is winning, it would take more than a lifetime to compute whether the *initial* position is winning. In fact the lifetime of our Galaxy would be too short for such a computation, even if we use the most powerful computer which we have at present. This phenomenon is known as *combinatorial explosion* and means that some functions are computable in theory but not in practice.

In other words, we would not bother about the computability of the *g* function or about the easiness of its computation. This means that the language which describes the world can even describe non-computable worlds. But, will our language be able to describe indescribable worlds? Can a world be indescribable? Yes, it can, but if a world is indescribable we would not be able to describe it. Whatever language we choose, the descriptions will be countably many but there will be lot more worlds than descriptions. So, there will obviously be indescribable worlds, but we will stick to the describable ones and assume that each indescribable world has a describable counterpart which is sufficiently similar to it.

The essence of the AI program

As we said before, we will reduce the task of creating AI to the task of finding a language which can describe worlds. If we had that descriptive language, how can we create the AI program by this language?

In [7] we wrote that AI should answer two questions: “What is going on?” and “What should I do?”. In this article we will not deal with the second question, but if we have understood the world, in our mind we can make a few steps forward and using an algorithm similar to Min-Max select the best action which is likely to bring about the best possible future development.

Answering the first question is more difficult. The first question is linked to understanding the world, i.e. to finding the *f* function which is the underlying essence of the world. Finding of the *f* function is tantamount to finding its description written in some language for description of worlds. As we said before, in addition to the function we have to find the current state of the world (the memory value). In order to find the current value, we will again need the language for description of worlds because that language will give us the format of the memory. In other words, before we find the current memory value, we must know the format in which the value is stored. (As we saw from the description of the *f* function, the memory format will be the group of questions *GQ* or more precisely the abstract version of the *GQ* group which describes only the number of questions and the number of answers to each question.)

If we have the language for description of worlds, we would know what we are looking for and will easily write a program which searches for such description. Of course, it will not be easy to make this search efficient.

The description must be discoverable. For this purpose it should consist of individual modules (individual questions). These modules must be patterns which can be discovered one by one. For example, if you try to guess my email password, you will have hard time because the possible combinations are far too many. Your task would be much easier if you were able to guest the password characters one by one. For the first character there are not too many choices and you can easily guess it, provided however that I am kind enough to tell you whether you have guessed it or not. Therefore, the description should consist of simple individual modules such that each module contains some telltale specificity by which it can be discovered.

How many agents will be there?

In every world there is at least one agent – the protagonist. This is the agent who receives inputs from the world and returns outputs which have an impact on the world. You are the protagonist in your world. You have a model of the world and your model describes many other agents. These agents are other humans as well as animals, deities and even inanimate objects because we expect some objects to perform certain actions from time to time. For example, your car breaks down. You might assume that the perpetrator of this action is the car itself. Conversely, when your car is in good order and running on the road you will not assume that it is driving on its free will, but this is exactly how dogs perceive the world. A dog barks at the car, not at the driver. I.e. the dog perceives the car as a separate agent and does not realize that it is driver inside who runs the car.

You may assume that in your world there is just one agent and that agent is you. As concerns all other people, you may assume that they are just a product of your imagination rather than real beings. Each world can be presented in a single-agent or multi-agent model, however the multi-agent model is a far more natural and understandable. This is the reason why most of us believe that they are not alone in the world and that in the world there are other people (agents) as well.

The particular world we will look at is the chess game and we will explore two versions of that world. In the first version there will be a single agent playing against himself. In the second version there will be two agents. The protagonist will play with the white pieces and there will be a second agent (antagonist). The second version is more interesting and sophisticated, but also more difficult to describe.

Why a particular world?

When we develop a new programming language, we do not create it in one go but first choose a particular problem and write a program which solves that problem. Then we pick another problem and write another program. Thus, we improve the language by tailoring it to the problems we need to solve.

We will take a similar approach in our quest for a language which describes worlds. We will take a particular world and will try to describe it in some language. Initially, that language will not be universal and capable of describing any world, but it would be a success if the language manages to describe the first world. We will gradually improve that language by applying it to various worlds until we eventually end up with the universal language we need.

The first world we start from is the chess game. More precisely, this will be a world in which the agent plays chess. Will the agent be solitaire in this world or will the world include another agent – an antagonist (opponent) who moves the black pieces?

We will start with the simpler version where the agent (the protagonist) plays solitaire by moving both the white pieces and the black pieces. Once the agent plays a white piece he will change sides and will play a black piece next. This is how people play when they are quarantined, jailed or placed in other isolation. Playing against an opponent is far more interesting, but that makes the world much more complicated. In the two-players case the world becomes non-computable because we will have an agent who plays black pieces and this agent executes an algorithm.

In the single-player case the world will be computable except for one rule: we cannot play a move after which we will be in check. This means we cannot play a move if it enables the opponent to capture our king in the next move. That is, if there is an algorithm by which the opponent can capture our king. (More precisely, the algorithm for moving the pieces is given, but there is a specific execution of this algorithm in which our opponent captures our king.)

Now I can hear you saying that this is perfectly computable so that if I tell you a move and a position, you will tell me right away whether you will be in check after that move. Correct, this is computable with chess, because everything about chess is finite (a finite 8 x 8 board). In the general case however the existence of an algorithm is non-computable (i.e. the existence of a specific execution of a given algorithm is non-computable.)

Which particular world?

Let the agent play chess by moving chess pieces on a chessboard. This world is emulated in the computer program [8] written in language [9].

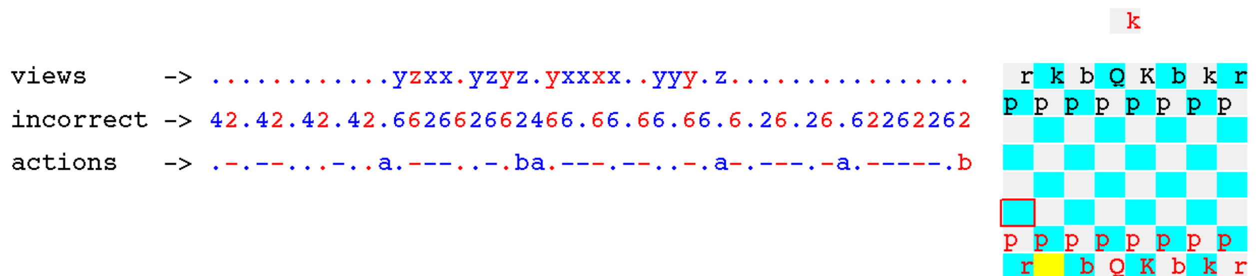

Figure 1

The left-hand side of Figure 1 shows the stream of input-output information, in fact not the full stream, but only the last 50 steps. The top row shows the agent's observations and the bottom row – the agent's actions. There are four possible observations: {0, x, y, z}. The possible actions are also four: {0, a, b, c}. For the sake of legibility the nil and the 'c' character are replaced by dots and minuses. In the second row we can see which actions are permitted at the present moment (2 means that action ***a*** is incorrect, 4 means that ***b*** is incorrect and 6 means that both ***a*** and ***b*** are incorrect).

All the agent can see is the left-hand side of Figure 1. The agent cannot see what is in the right-hand side, and must figure it out in order to understand the world. In the right-hand side we can see (i) the position on the board, (ii) the piece lifted by the agent (knight), (iii) the place from which the knight was lifted (the yellow square) and the square observed currently (the one framed in red).

Partial Observability

We have restricted the agent's observability to a single square (the currently observed one) rather than letting him see the full chessboard. The agent can move his gaze from one square to another and thus obtain an overview of the full board. Nevertheless, the important assumption here is that the agent cannot observe the entire board so we have Partial Observability. This requires the agent to imagine the part of the world which he cannot see at the moment. All the agent can see is a single square, therefore the rest of the chessboard will be in his imagination.

The agent's imagination must cater on some idea about the current state of the world (i.e. about s_i). What sort of idea? In the best-case scenario, the agent should know the current state of the world with utmost precision (i.e. should have a deterministic answer to all questions asked). If utmost precision is not achievable, the agent should have some approximated knowledge of the state of the world (deterministic answers to certain questions and non-deterministic answers to all other questions, e.g. several possibilities or "Do not know"). When the agent moves a piece on the board and his virtual opponent makes a countermove, in the first instance that countermove would be unknown to the agent. What the agent can do is obtain an overview of the entire chessboard (by moving the currently observed square) and figure out what the opponent's last move has been. Thus, from a non-deterministic answer the agent can arrive at a deterministic one.

We use coding

The agent will be able to do 8 things: move his gaze (the square currently observed) in the four directions, lift the piece he sees at the moment and drop the lifted piece in the square he sees at this moment. The seventh and eighth thing the agent can do is do nothing.

We will limit the agent's actions to the four characters {0, a, b, c}. The 0 and 'c' symbols will be reserved for the "do nothing" action. This leaves us with 6 actions to describe with as little as 2 characters. How can we do that? We will do that by coding: Let us divide the process in three steps. Every first step will describe how we move the square in horizontal direction (i.e. how we move the observation gauge). Every second step will describe how we move the square in vertical direction and every third step will indicate whether we lift a piece or drop the lifted piece.

We mentioned in [5] that we should avoid excessive coding because the world is complicated enough and we do not want to complicate it further. However, out coding here is not excessive because it replaces eight actions with four and therefore simplifies the world rather than complicate it.

Two void actions

Why do we introduce two actions that mean "I do nothing"? Actually, when the agent just stays and does nothing, he observes the world. The question is, will he be a passive observer or he will observe actively?

When you just stay and observe the world, you are not a passive observer. At the very least, you are moving your gaze.

All patterns that the passive observer can see are periodic. In a sense, the periodic patterns are few and not very interesting. Much more interesting are the patterns that the active observer can see.

We expect the agent to be able to notice certain patterns (properties). For example, the type and color of pieces are such properties. When the agent stays in a square and does nothing, it will be difficult for him to detect the pattern (property), especially since he may have to detect two or three patterns at the same time. If the agent is active and can alternate two actions, then the patterns he observes will be much clearer and more quickly detectable.

To distinguish between the two "I do nothing" actions, we called the second one "surveillance".

1, 2, 3

The first pattern which will exist in this particular world stems from our division of the steps in three groups. Let us name this pattern "1, 2, 3". The pattern is modeled in Figure 2.

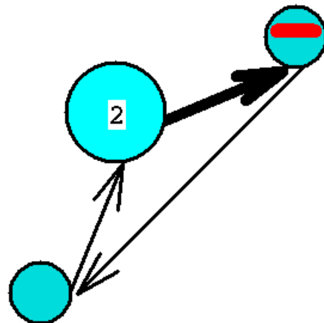

Figure 2

What is the gist of this pattern? It counts: one, two, three.

The pattern is presented through an Event-Driven (ED) model. (ED models were first introduced in [3, 4] and elaborated in [6]). This particular ED model has three states. The event in this model is only one and this is the event "always" (i.e. "true" or "at each step").

This ED model corresponds to a question with three basic answers. The question is "In which state is the model now?". The number of basic answers to this question is equal to the number of states because the model is deterministic. If the model were non-deterministic, the basic answers would be as many as the subsets of states of the model.

Trace

Does anything specific occur in the states of the above model so that we can notice it and thereby discover the model? In other words, is there a "trace" (this terminology was introduced in [6])?

Yes, in the third state, action ***a*** or action ***b*** (or both) must be incorrect. The reason is that the third state indicates whether we lift the piece we see or drop a piece which is already lifted. These two actions cannot be possible concurrently.

We can describe the world without this trace, but without it the "1, 2, 3" pattern would be far more difficult to discover. That is why it is helpful to have some trace in this model.

The trace is the telltale characteristic which makes the model meaningful. Example: cold beer in the refrigerator. Cold beer is what makes the fridge a more special cupboard. If there was cold beer in all cupboards, the refrigerator would not be any special and it would not matter which cupboard we are going open.

The trace enables us predict what is going to happen. When we open the fridge, we expect to find cold beer inside. Furthermore, the trace helps us recognize which state we are in now and thereby reduce non-determinacy. Let us open a white cupboard, without knowing whether it is the fridge or another white cupboard. If we find cold beer inside, then we will know that we have opened the fridge and thus we will reduce non-determinacy.

We will consider two types of traces – permanent and moving. The permanent trace will be the special features (phenomena) which occur every time while the moving trace will represent features which occur from time to time (transiently).

An example in this respect is a house which we describe as an Event-Driven model. The rooms will be the states of that model. A permanent feature of those rooms will be number of doors. Transient phenomena which appear and then disappear are "sunlit" and "warm". I.e. the permanent trace can tell us which room is actually a hallway between rooms and the moving trace will indicate which room is warm at the moment.

Rooms can be linked to various objects. These objects have properties (the phenomena we see when we observe the relevant object). Objects can also be permanent or moving and accordingly their properties will be permanent or transient phenomena. Furniture items (in particular heavyweight ones) are examples of permanent objects. People and pets are examples of moving objects. To sum up, a fixed trace will describe what is permanent and a moving trace will describe what is transient.

We will associate the permanent trace with a question which always returns the same answer. We will not deal with these questions since we assume that each question has more than one possible answer. We will also assume that permanent questions are not part of the state of the world and are instead inbuilt in the definition of the g function. Nevertheless, as we improve the g function and change our idea of the world, we will assume that some permanent question may evolve into a real question. For example, "How many doors are there in this room?" is a permanent one until some enthusiast drills another door.

We will associate the moving trace with a question such as "Is the cat in the room?". This question has two answers which depict the current state of the world. The question "How many cats are there in the room?" has more answers the number of which depends on the number of cats we have in the house. If another cat comes by, the model of the world will change and so will the number of answers to this question.

Horizontal and Vertical

The next Event-Driven model we need for our description of the world is the Horizontal model (Figure 3).

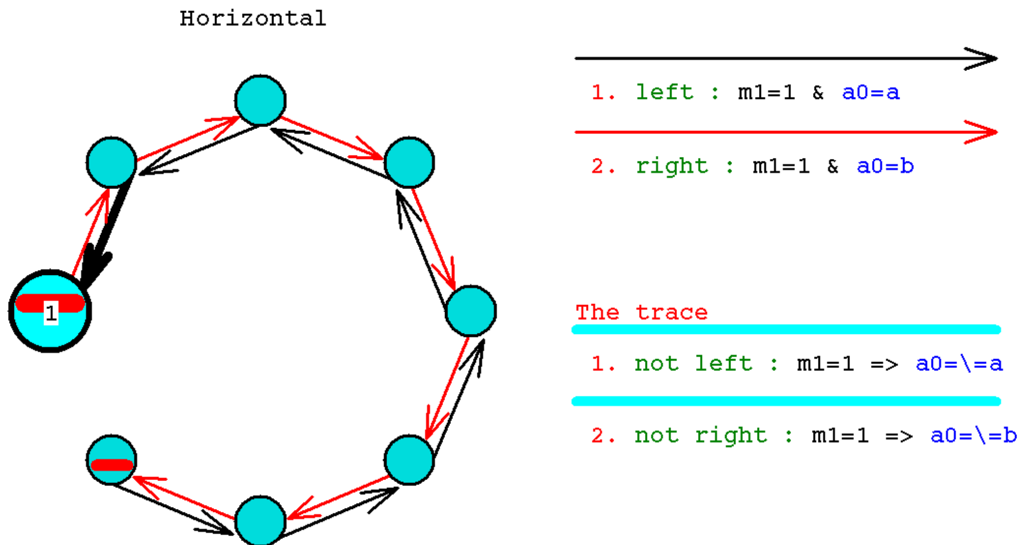

Figure 3

This model tells us in which column of the chessboard is the currently observed square.

Here we have two events: *left* and *right* which reflect the direction in which the agent moves his gaze – to the left or to the right. So, the agent performs the actions ***a*** and ***b*** when model 1 is in state 1. We also have two traces. In state 1 playing to the left is not possible. Therefore, the *left* event cannot occur in state 1. Similarly, we have state 8 and the trace that playing to the *right* is not possible. These two traces will make the model discoverable. For example, if you are in a dark room which is 7 paces wide, you will find that after making 7 paces you cannot continue to the left. You will realize this because you will bump against the wall. Therefore, the trace in this case will be the bump against the wall. These bumps will occur only in the first and in the last position.

In addition to helping us discover the model, the trace will do a nice job explaining the world. How else would you explain that in the leftmost column one cannot play *left*.

Very similar is the Vertical model shown on Figure 4.

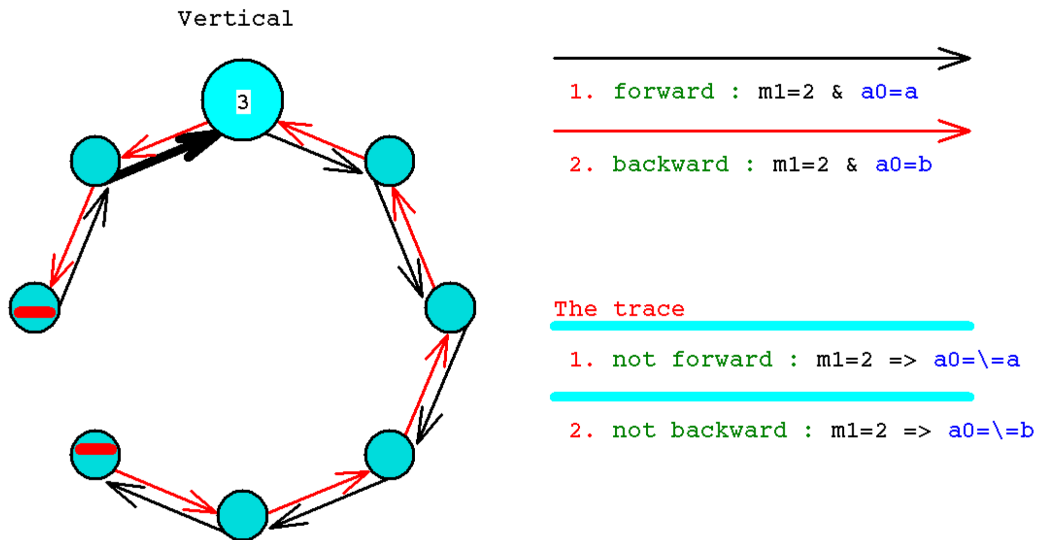

Figure 4

This model will tell us the row of the currently observed square. Likewise, we have two events (*forward* and *backward*) and two traces (*forward move not allowed* and *backward move not allowed*).

It makes perfect sense to do the Cartesian product of the two models above and obtain a model with 64 states which represents the chessboard.

The bad news is that our Cartesian product will not have a permanent trace. In other words, nothing special will happen in any of the squares. Indeed, various things happen, but they are all transient, not permanent. For example, seeing a white pawn in the square may be relatively permanent, but not fully permanent, because the player can move the pawn at some point of time.

Thus we arrive at the conclusion that the trace may not always be permanent.

The moving trace

As we said, moving traces are the special features which occur in a given state only from time to time (transiently), but not permanently.

How can we depict a moving trace? In the case of permanent traces, we indicated on the state whether an event occurs always in that state (by using red color and accordingly blue color for events which never occur in that state).

We will depict the moving trace by an array with as many cells as are the states in the model under consideration. In each cell we will write the moving traces which are in the corresponding state. That is, the moving trace array will be changing its values.

Here is the moving trace array of the Cartesian product of models 2 and 3:

8	black rook unmov	black knight	black bishop	black queen	black king unmov	black bishop	black knight	black rook unmov
7	black pawn unmov	black pawn unmov	black pawn unmov	black pawn unmov	black pawn unmov	black pawn unmov	black pawn unmov	black pawn unmov
6								
5								
4								
3								
2	white pawn unmov	white pawn unmov	white pawn unmov	white pawn unmov	white pawn unmov	white pawn unmov	white pawn unmov	white pawn unmov
1	white rook unmov	lift	white bishop	white queen	white king unmov	white bishop	white knight	white rook unmov
	1	2	3	4	5	6	7	8

Figure 5

This moving trace is very complicated because it pertains to a model with 64 states. Let us take the moving trace of a model with two states (Figure 6). This is model 4 which remembers whether we have lifted a chess piece. Its moving trace will remember which the lifted piece is. Certainly, the model will also have its permanent trace which says that in state 2 *lifting piece is impossible* and in state 1 *dropping piece is impossible*.

The moving trace of that model will be an array with two cells which correspond to the two states of the ED model. The cell which corresponds to the current state is framed in red. The content of the red cell is not very important. What is important is the content of the other cells because they tell us what will happen if one of these other cells becomes the current cell. In this case, if we drop a lifted piece we will go to state 1, where we will see the lifted piece. (We will see the piece which we have dropped, in this case a white knight.)

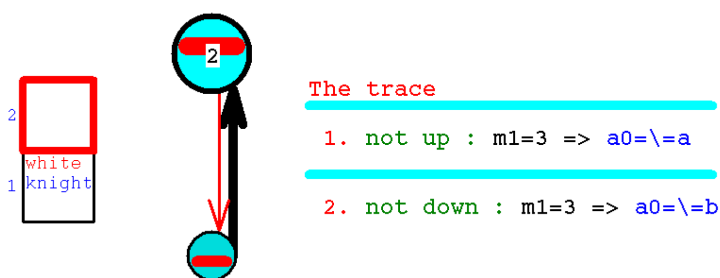

Figure 6

We said that the language for description of worlds will tell us how the memory of the *f* function looks like. Where is the internal state of the world stored? At two locations – first, the current

state of each ED model and second, the moving traces. For example, in Figure 5 we can see how the moving trace presents the position on the chessboard.

If the language for description of worlds were a standard programming language, its memory would hold the values of the variables and of the arrays. By analogy, we can say that the current state of the ED model is the value of one variable and the value of one moving trace is the value of one array.

The value of the current state of an ED model is usually a number when the model is deterministic or several numbers if the ED model has several current states. The value of each cell of the moving trace array will consist of several numbers because one state can have many moving traces. Certainly, the permanent traces can also be more than one.

If we present the current state of the world by means of questions, we will have one question for the current state of each ED model and one question for each cell of each moving trace. The number of answers to these questions depends on the number of states of the ED model and on whether the model is deterministic, as well as on the number of moving traces that can be present in the corresponding moving trace cell.

Algorithms

Now that we have described the basic rules of the chess game and the position of the chessboard, the next step is to describe how the chess pieces move. For this purpose we will resort to the concept of algorithm.

For most people an algorithm is a Turing machine. The reason is that they only look at $\mathbb{N} \rightarrow \mathbb{N}$ functions and see the algorithm as something which computes these functions. To us, an algorithm will describe a sequence of actions in an arbitrary world. In our understanding, algorithms include cooking recipes, dancing steps, catching a ball and so forth. We just said a sequence of actions. Let us put it better and change this to a sequence of events. An action is an event, but not every event is an action or at least our action – it can be the action of another agent. The description of the algorithm will include our actions as well as other events. For example, we wait for the water in our cooking pan to boil up. The boiling of water is an event which is not our action.

In our definition, an algorithm can be executed without our participation at all. The Moonlight Sonata, for example, is an algorithm which we can execute by playing it. However, if somebody else plays the Moonlight Sonata, it will still be an algorithm albeit executed by someone else. When we hear the piece and recognize that it is the Moonlight Sonata, we would have recognized the algorithm even though we do not execute it ourselves.

Who actually executes the algorithm will not be a very important issue. It makes sense to have somebody demonstrate the algorithm to us first before we execute it on our own.

We will examine three versions of algorithm:
1. Railway track;
2. Mountain footpath;
3. Going home.

In the first version there will be restrictions which do not allow us to deviate from the execution of the algorithm. For example, when we board a coach, all we can do is travel the route. We cannot make detours because someone else is driving the coach. Similarly, when listening to someone else's performance of the Moonlight Sonata, we are unable to change anything because we are not playing it.

In the second version, we are allowed to make detours but then consequences will occur. A mountain footpath passes near an abyss. If we go astray of the footpath we will fall in the abyss.

In the third version, we can detour from the road. After the detour we can go back to the road or take another road. The Going home algorithm tells us that if we execute it properly, we will end up at our home, but we are not anyhow bound to execute it or execute in exactly the same way.

Typically, we associate algorithms with determinacy. We picture in our mind a computer program where the next action is perfectly known. However, even computer programs are not single-threaded anymore. With multi-threaded programs it is not very clear what the next action will be. Cooking recipes are even a better example. When making pancakes, we are not told which ingredient to put first – eggs or milk. In both cases we will be executing the same algorithm.

Imagine an algorithm as a walk in a cave. You can go forward, but you can also turn around and go backward. The gallery has branches and you are free to choose which branch to take. Only when you exit the cave you will have ended the execution of the cave walk algorithm. In other words, we imagine the algorithm as an oriented graph with multiple branches and not as a road without any furcations.

The algorithm of chess pieces

We will use algorithms to describe the movement of chess pieces. We will choose the Railway track version (the first one of the versions examined above). This means that when you lift a piece you will invoke an algorithm which prevents you from making an incorrect move.

We could have chosen the Mountain footpath which allows you to detour from the algorithm, but with consequences. For example, lift the piece and continue with the algorithm, but if you break it at some point the piece will escape and go back to its original square.

We could have chosen the Going home version where you can move as you like but can drop the piece only at places where the algorithm would put it if it were properly executed. That is, you have full freedom of movement while the algorithm will tell you which moves are the correct ones.

We will choose the first version of the algorithm mainly because we have let the agent play randomly and if we do not usher him in some rail track, he will struggle a lot in order to make a correct move. Moreover, we should consider how the agent would understand the world. How would he discover these algorithms? If we put him on a rail track, he will learn the algorithm – like it or not – but if we let him loose he would have hard time trying to guess what the rules for movement (the algorithms) are. For example, if you demonstrate to a school student the algorithm of finding a square root, he will learn to do so relatively easily. But, the kid's life would become very difficult if you just explain to him what is square root and tell him to find the

algorithm which computes square roots. You can show the student what a square root is with a definition or examples, but he would grasp it more readily if you demonstrate hands-on how the algorithm works.

What will be the gist of our algorithms? These will be Event-Driven models. There will be some input event which triggers the algorithm and another output event which will put an end to the execution of the algorithm. Later on we will clone the outputs in two (successful and unsuccessful output).

Each chess piece will have its algorithm:

The king and knight algorithms

The king's algorithm (Figure 7) will be the simplest one. The input event will be *king lifted*. The input point will be state 1 (this applies to all algorithms described here). The events will be four (*left, right, forward* and *backward*).

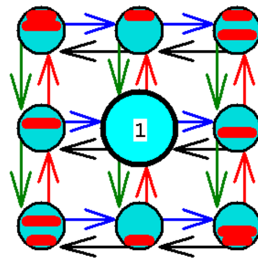

Figure 7

The trace will consist of four events (*left move not allowed*, *right move not allowed*, etc.). These four events (traces) will restrict the king's movement to nine squares. Thus, the four events (traces) will be the rail tracks in which we will enter and which will not let us leave the nine squares until we execute the algorithm. In Figure 7, the four traces are marked with red horizontal lines. For example, the three upper states have the first trace which means that the king cannot move forward from these three states.

We may drop the lifted piece (the king) whenever we wish. Certainly, there will be other restrictive rules and algorithms. E.g. *we cannot capture our own pieces* is an example of other restrictions, which however are not imposed by this algorithm. If we drop the piece in state 1, the move will not be real but fake. If we drop the piece in another state, then we would have played a real move.

The knight's algorithm is somewhat more complicated (Figure 8). The main difference with the king's algorithm is that here we have one more trace. This trace restricts us such that in certain states we cannot drop the lifted piece. (Only this trace is marked in Figure 8, the other four traces are not.) In this algorithm we have only two options – play a correct move with the knight or play a fake move by returning the knight to the square which we lifted it from.

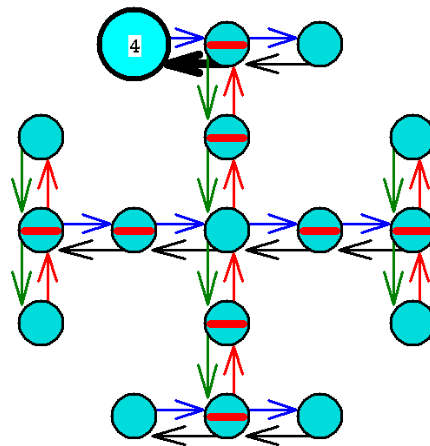

Figure 8

The rook and bishop algorithms

Although with less states, the rook's algorithm is more complex (Figure 9). The reason is that this algorithm is non-deterministic. In state 3 for example there two arrows for the *move forward* event. Therefore, two states are candidates to be the next state. This non-determinacy is resolved immediately because in state 1 it must be seen that a piece has been lifted from that square while the opposite must be seen in state 3. Therefore, we have a trace which resolves the algorithm's non-determinacy immediately.

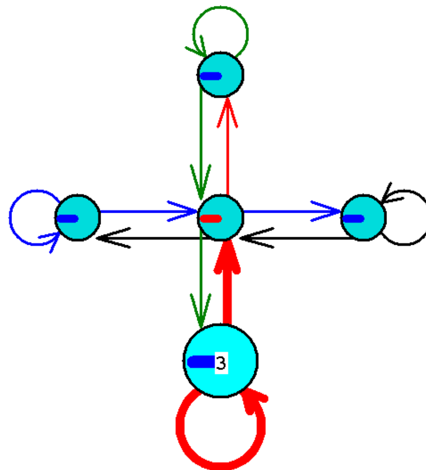

Figure 9

Even more complicated is the bishop's algorithm (Figure 10). The reason is that we cannot move the bishop diagonally outright and have to do this in two steps: first a horizontal move and then a vertical move. If the event *left* occurs in state 1, we cannot know whether our diagonal move is *left and forward* or *left and backward*. This is another non-determinacy which cannot however be resolved immediately. Nevertheless, the non-determinacy will be resolved when a *forward* or *backward* event occurs. In these two possible states we have traces which tell us "*forward move not allowed"* in state 8 and "*backward move not allowed"* in state 2. If the *no-forward* restriction applied in both states, the *forward* event would breach the algorithm. But in this case the event is allowed in one of the states and disallowed in the other state. So, the *forward* event is allowed, but if it occurs state 8 will become inactive and the non-determinacy will be resolved. (In Figure 10 we have marked only the *no-forward* and the *no-backward traces*.)

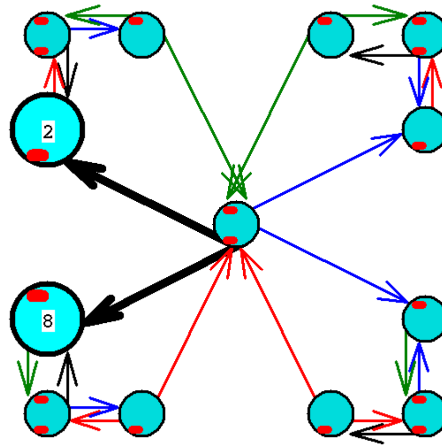

Figure 10

The most complicated algorithm is that of the queen because it is a combination of the rook and bishop algorithms. The pawn's algorithm is not complicated, but in fact we have four algorithms: for white/black pawns and for moved/unmoved pawns.

The term *algorithm*

Importantly, this article defines the term *algorithm* as such. Very few people bother to ask what is an algorithm in the first place. The only attempts at a definition I am aware of are those of Moschovakis [1, 2]. In these works Moschovakis says that most authors define algorithms through some abstract machine and equate algorithms with the programs of that abstract machine. Moschovakis goes on to explain what kind of an algorithm definition we need – a generic concept which does not depend on a particular abstract machine. The computable function is such a concept, but for Moschovakis it is too general so he seeks to narrow it down to a more specific concept which reflects the notion that a computable function can be computed by a variety of substantially different algorithms. This is a tall aim which Moschovakis could not reach in [1]. What he did there can be regarded as a new abstract machine. Indeed, the machine is very interesting and more abstract than most known machines, but again we run into the trap that the machine's program may become needlessly complicated and in this way morph into a new program which implements the same algorithm. Although [1] does not achieve the objective of creating a generic definition of an algorithm, Moschovakis himself admits that his primary objective is to put the question on the table even if he may not be able to answer it. His exact words are: "my chief goal is to convince the reader that the problem of founding the theory of algorithms is important, and that it is ripe for solution."

The Turing machine

So far we described the chess pieces algorithms as Event-Driven (ED) models. Should we assert that each algorithm can be presented as an ED model? Can the Turing machine be presented in this way?

We will describe a world which represents the Turing machine. The first thing we need to describe in this world is the infinite tape. In the chess game, we described the chessboard as the moving trace of some ED model with 64 states. Here we will also use a moving trace, however

we will need a model with countably many states. Let us take the model in Figure 3. This is a model of a tape comprised of eight cells. We need the same model which has again two events (*left* and *right*), but is not limited to a *leftmost* and *rightmost* state. This means an ED model with infinitely many states. So far we have only used models with finitely many states. Now we will have to add some infinite ED models which nevertheless have structures as simple as this one. In this case the model is merely a counter, which keeps an integer number (i.e. an element of $\mathbb{Z}$). The counter will have two operations (*minus one* and *plus one*) or (*move left* and *move right*). The addition of an infinite counter expands the language, but as we said we will keep expanding the language in order to cover the worlds we aim to describe.

What kind of memory will this world have? We must memorize the counter value (that is the cell on which the head of machine is placed). This is an integer number. Thus, we will have a question with a countable number of answers. Besides this, we will need to memorize what is stored on the tape. For this purpose we will need an infinite sequence of 0 and 1 numbers, which is equinumerous to the continuum. We will express this by countably many questions, each of which has two answers. We usually use Turing machines in order to compute $\mathbb{N} \rightarrow \mathbb{N}$ functions. In this case we can live only with configurations which use only a finite portion of the tape, i.e. we can consider only the describable answers to the questions. Nevertheless, all configurations of the tape are continuum many and all answers to the questions asked are continuum many.

Note: The agent's idea of the state of the world will be countable even though the memory of the world is a continuum. In other words, the agent cannot figure out all possible configurations on the tape, but only a countable subset of these configurations. In this statement we imagine the agent as an abstract machine with an infinite memory. If we image the agent as a real computer with a finite memory, in the above statement we must replace *countable* with *finite*. Anyway, if the agent is a program for a real computer, the finite memory would be enormous, so for the sake of simplicity we will deem it as countable.

Thus, we have described the tape of the Turing machine with an infinite ED model. In order to describe the head of the machine (the algorithm *per se*) we will need another ED model. We will employ the Turing machine in order to construct the second ED model.

We assumed that the machine uses two characters {0, 1}. Let us construct an ED model with four events:
write(0),
write(1),
move left,
move right.

Then each command to the machine will be in the following format:

if observe(0) then write Symbol_0, move Direction_0, goto Command_0
if observe(1) then write Symbol_1, move Direction_1, goto Command_1

Here Symbol_i, Direction_i and Command_i have been replaced with concrete values. For example:
if observe(0) then write(1), move left, goto s_3
if observe(1) then write(0), move right, goto s_7

We will replace each command with four states which describe it. The above command will take the following form:

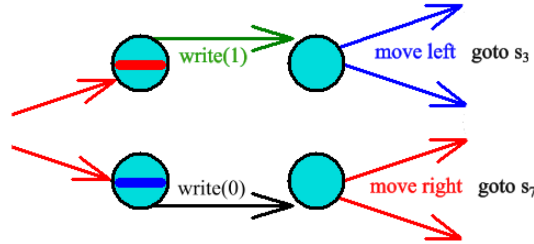

Figure 11

In Figure 11, the input is over the *move right* event. In fact there will be many input paths – sometimes over the *move left* event and other times over the *move right* event. Importantly, the input will be non-deterministic but the non-determinacy will be resolved immediately because the first two states have a trace. In the top state the event "observe(0)" must always occur and in the bottom state the "observe(0)" event must never occur.

Thus, each state of the machine is replaced with four states as shown in Figure 11 and then the individual quaternaries are interconnected. For example, the quaternary in Figure 11 connects to the quaternary in s_3 by arrows over the *move left* event and to the quaternary in s_7 by arrows over the *move right* event.

We have to add some more trace to accommodate the rule that only one of the four events is possible in each state. The new trace should tell us that the other three events are impossible. We should do this in case we want a Railway track type of algorithm. If we prefer a Mountain footpath algorithm, we must add a trace which tells which consequences will occur if one of the other three events happens. If we wish a Go home algorithm, then the other three events must lead to a termination of the algorithm.

Thus we presented the Turing machine through an Event-Driven model or, more precisely, through two ED models – the first one with infinitely many states and the second one with a finite number of states (the number of machine states multiplied by 4).

Who executes the algorithm of the Turing machine? We may assume the four events are actions of the agent and that he is the one who runs the algorithm. We may also assume that the events are acts of another agent or that the events just happen. In that case the agent will not be the executor of the algorithm, but just an observer. In the general case, some events in the algorithm will be driven by the agent and all the rest will not. For example, "I pour water in the pot" is an action of the agent while "The water boils up" is not his action. The agent can influence even those events which are not driven by his actions. This is described in [7]. For these events the agent may have some "preference" and by his "preferences" the agent could have some influence on whether an event will or will not occur.

Properties

Having defined the term *algorithm*, we will try to define another fundamental term: *property*.

For the definition of this term we will again resort to the Event-Driven models. A property is the phenomenon we see when we observe an object which possesses that property. Properties are patterns which are not observed all the times but only from time to time. Given that the other patterns are presented through ED models, it makes perfect sense to present properties through ED models, too.

The difference between a pattern and a property will be that the pattern will be active on a permanent basis (will be observed all the time) while the property will be observed from time to time (when we observe the object which possesses that property).

The basic term will be *property* while *object* will be an abstraction of higher order. For example, if in the chess world one observes the properties *white* and *knight* he may conclude that there is a *white knight* object which is observed and which possesses these two properties. We may also dispense of this abstraction and simply imagine that some properties come and go, i.e. some phenomena appear and disappear.

The second coding

The agent's output consists of four characters only which made us use coding in other to describe the eight possible actions of the agent. The input is also limited to four characters. While it is true that the input will depict to us only one square rather than the full chessboard, four characters are still too little because a square can accommodate six different pieces in two distinct colors. Furthermore, we need to know whether the pawn on the square has moved and whether the lifted piece comes from that square. How can one present all that amount of information with four characters only?

That information may not necessarily come to the agent for one step only. The agent can spend some time staying on the square and observing the input. As the agent observes the square, he may spot various patterns. The presence or absence of each of these patters will be the information which the agent will receive for the square he observes. Although the input characters are only four, the patterns that can be described with four characters are countless.

Let us call these patterns *properties* and assume that the agent is able to identify (capture) these patterns. We will further assume that the agent can capture two or more patterns even if they are layered on top of each other. Thus, the agent should be able to capture the properties *white* and *knight* even when these properties appear at the same time.

How would the properties look like? In the case of chess pieces, the patterns and algorithms of their movement are written by a human who has an idea of the chess rules and of how the pieces move. The properties are not written by a human and are generated automatically. As an example, Figure 12 depicts the property *pawn*. That property appears rather bizarre and illogical. The reason is, as we said, that the property is generated automatically in a random way. It is not written by us because we do not know how the pawn would look like. How the pawn looks like does not matter. What matters is that the pawn should have a certain appearance such that it can be recognized by the agent. In other words, the pawn should have some face, but how that face would look like is irrelevant.

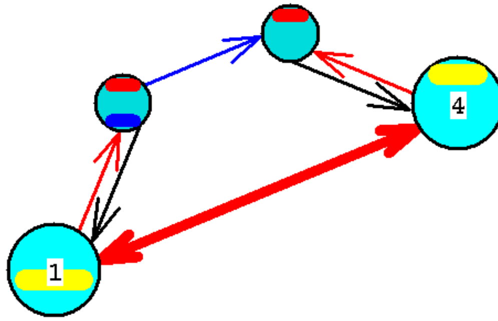

Figure 12

In our program [8] there are 10 properties and each of them has some trace. When several properties are active at the same time, each of them has an impact (via its trace) on the agent's input. Sometimes these impacts may be contradictory. For example, one property tells us that the next input must be the character ***x***, while another property insists on the opposite (the input must not be ***x***). The issue at hand is then solved by voting. The world counts the votes for each decision and selects the one which reaps the highest number of votes. As concerns contradictory recommendations, they will cancel each other.

A deterministic world

So far we have described a chess world where the agent plays solitaire against himself. The description consists of 24 modules (Event-Driven models), two arrays (moving traces) and 7 additional rules. These rules provide us with additional information about how the world has changed. For example, when we lift a piece, the property *Lifted* will appear in the square of that piece. We are able to formulate these rules owing to the fact that we have the context of the chessboard (the moving trace from Figure 5). If we had no idea that a chessboard exists, we would not be able to formulate rules for the behavior of the pieces on that board. We wrote a computer program [8] which emulates this world. In that demonstration program the agent plays randomly. Of course the agent's actions are not important. What matters is that we have described the *g* function. Therefore, we have described the world.

The description thus obtained is deterministic, i.e. the initial answer is deterministic and the *g* function is also deterministic. (Meaning that for each deterministic answer and for each action there will be one observation, for which the *g* function will return a deterministic answer and a probability of 1. For all other observations the *g* function will return a probability of 0.)

A deterministic description means that the described world is free from randomness. Should the description of the world be deterministic? Should we deal only with deterministic descriptions? The idea that the world may be deterministic seems outlandish. And even if it were, we need not constrain ourselves to deterministic descriptions.

If we apply a deterministic description to a non-deterministic world, that description will very soon exhibit its imperfections. Conversely, the world may be deterministic but its determinacy may be too complicated and therefore beyond our understanding (rendering us unable to describe it). Accordingly, instead of a deterministic description of the world will find a non-deterministic description which works sufficiently well.

Typically, the world is non-deterministic. When we shoot at a target may miss it. This means that our actions may not necessarily yield a result or may yield different results at different times.

We will assume that the *f* function may be non-deterministic. For most authors, non-deterministic implies that for each possible value of the *f* function there is one precisely defined probability. In [6] and [7] we showed that the latter statement is too deterministic. Telling the exact probability of occurrence for each and every event would be an exaggerated requirement. Accordingly, we will assume that we do not know the exact probability, but only the interval *[a, b]* in which this probability resides. Typically that would be the interval *[0, 1]* which means that we are in total darkness as regards the probability of the event to occur.

We will sophisticate the chess game by adding another agent: the opponent (antagonist) who will play the black pieces. This will induce non-determinacy because we will not be able to tell what move the opponent will play. Even if the opponent is deterministic, his determinacy would be too complicated for us to describe it. That is why will describe the world through a non-deterministic function *g*.

Random events

One way of introducing non-determinacy in Event-Driven models is by random events. If all events in the ED model are deterministic, the ED itself behaves deterministically (even if the model is non-deterministic, the set of its active states is determinated). To get the ED behave as a non-deterministic one, we need to add random events in it. All we know about random events is that they occur with a certain probability, so we do not know when they exactly occur. (The probability is a number or interval.)

While random events are not used in either of the two worlds examined in this article, we do mention them here because they will be needed in other worlds. When seeking a model of the world we can perfectly assume that we have discovered some event which switches the states of some ED model. Suppose we have discovered that event indirectly (as described in [7]) meaning that we can observe the event but do not know its characteristic function. Then it makes sense to assume that it is a random event which occurs with a certain probability. Later on we can find out when exactly the event occurs whereupon the event will become deterministic (i.e. later on we can describe it by a characteristic function).

Impossible events

We said the world would be more interesting if we do not pay against ourselves but against another agent who moves the black pieces.

For this purpose we will modify the fifth ED model (the one which tells us which pieces we are playing with – white or black). This model has two states which are switched by the event *change*. The event is defined as “real_move” (this is the event when we make a real move while in “fake_move” we only touch a piece). We will change the definition of that event and define it as *never* (this is the opposite of *every time*).

Does it make sense to describe in our model events which can never happen? The answer is yes, because these events may happen in our imagination. I.e. even though these events not occur, we need them in order to understand the world. For example, we are unable to fly or change our gender, but we can do this in our imagination. The example is not very appropriate, because we

can already fly and change our gender if we wish to. In other words, we can imagine impossible events in our minds. Also, at some point, these impossible events may become possible.

We will use the impossible event *change* in order to add the rule that we cannot play a move after which we will be in check (our opponent will be able to capture our king). Figure 13 depicts the algorithm which describes how we switch sides (turn the chessboard around) and capture the king. If this algorithm is possible, then the move is incorrect.

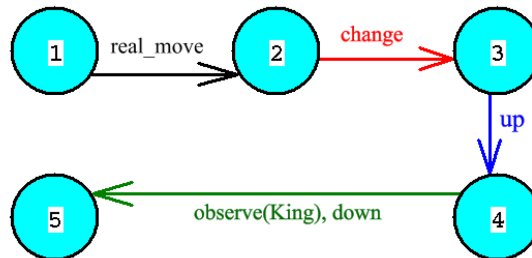

Figure 13

This algorithm is more consistent with our understanding of algorithms. While the algorithms of the chess pieces were oriented graphs with multiple branches, this one is a path without any branches. In other words, this algorithm is simply a sequence of actions without any diversions.

This algorithm needs some more restrictions (traces) which are not shown in Figure 13. In state 1 for example we cannot move in any of the four directions (otherwise we could go to another square and play another move). The *change* event can only occur in state 2 and not in any other state. In state 4 we have the restriction “not observe(King) => not down“ which means that the only move we can make is to capture the king.

Part of this algorithm is the impossible action *change*. As mentioned above, although this action is not possible, we can perform it in our imagination. This event may be part of the definition of algorithms which will not be executed but are still important because we need to know whether their executions exist.

Note: In this article, by saying that an algorithm can be executed we mean that it can be executed successfully. This means that the execution may finish in a final (accepting) state or with an output event (successful exit).

The second agent

The algorithm in Figure 13 would become simpler if we allow the existence of a second agent. Instead of switching between the color of the pieces (turning the chessboard around) we will replace the agent with someone who always plays the black pieces. Thus, we will end up with an algorithm performed by more than one agent, which is fine because these algorithms are natural. For example, “I gave some money to someone and he bought something with my money” is an example of an algorithm executed by two agents.

The important aspect here is that once we move a white piece, we will have somebody else (another agent) move a black piece. While in the solitaire version of the game we wanted to know whether a certain algorithm is possible, in the two players version we want a certain algorithm to be actually executed. Knowing that a certain algorithm is possible and the actual execution of that

algorithm are two different things. Knowing that “someone can cook pancakes” is okay but “your roommate cooking pancakes this morning” is something different. In the first case you will know something about the world while in the latter case you will have some pancakes for breakfast. If the actual execution of an algorithm will be in the hands of agent, then it does matter who the executing agent is. E.g. we will suppose that the pancakes coming from the your roommate’s hands will be better than those cooked by you.

We will assume that after each “real_move” we play, the black-pieces agent will execute the algorithm in Figure 14.

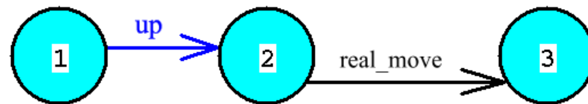

Figure 14

The execution of an algorithm does not happen outright because it is a multi-step process. Nevertheless, we will assume here that the opponent will play the black pieces right away (in one step). When people expect someone to do something, they tend to imagine the final result and ignore the fact that the process takes some time. Imagine that “Today is my birthday and my roommate will cook pancakes for me”. In this reflection you take the pancakes for granted and do not bother that cooking the pancakes takes some time.

As mentioned already, it matters a lot who the black-pieces agent is. Highly important is whether the agent is friend or foe (will he assist us or try to disrupt us). The agent’s smartness is also important (because he may intend certain things but may not be smart enough to do these things). It is also important to know what the agent can see. In the chess game we assume that the agent can see everything (the whole chessboard) but in other worlds the agent may only be able to know and see some part of the information. The agent’s location can also be important. Here will assume that it is not important, i.e. wherever the agent is, he may move to any square and lift the piece in that square. In another assumption the agent’s position may matter because the pieces that are nearer the agent may be more likely to be moved than the more distant pieces.

Agent-specific state

We assumed here that the second agent sees a distinct state of the world. I.e. the second agent has his distinct position $\langle x, y \rangle$ on the chessboard (the square which he observes). We also assume that the second agent plays with black pieces as opposed to the protagonist playing with the white pieces.

We assume that the two agents change the world through the same *g* function but the memory of that function is specific to each agent. (The function’s memory describes the state of the world through answers to the questions asked.) We might assume that the two states of the world have nothing in common, but then the antagonist’s actions will not have any impact on the protagonist’s world. Therefore, we will assume that the chessboard position is the same for both agents (i.e. the trace in Figure 5 is the same for both). We will further assume that each agent has his own coordinates and a specific color for his pieces (i.e. the active states of Event-Driven models 2, 3 and 5 are different for the two agents). As concerns the other ED models and the trace in Figure 6, we will assume that they are also specific to each agent, although nothing prevents us from making the opposite assumption.

Had we assumed that the two agents share the same state of the world, the algorithm in Figure 14 would become heavily complicated. The antagonist would first turn the chessboard around (*change*), then play his move and then turn the chessboard around again in order to leave the protagonist's world unchanged. Moreover, the antagonist would need to go back to his starting coordinates <*x, y*> (these are the protagonist's coordinates). It would be bizarre to think that the separate agents are absolutely identical and share the same location. The natural way of thinking is that the agents are distinct and have distinct, but partially overlapping states of the world. For example, "Right now I am cooking pancakes and my roommate is cooking pancakes, too". We may be cooking the same pancakes or it may be that my pancakes have nothing to with his.

Note: It is not very accurate to say that the world has distinct states for the agents. The world is one and it has only one state. It would be more accurate to say that we have changed the world and now we have a world with more questions. The questions that are common to the two agents have remained the same, but the other questions are now bifurcated. For example, "Where am I?" is replaced with the questions "Where is the protagonist?" and "Where is the antagonist?". From the *g* function we derived the new function g' which operates with the answers to the new questions and describes (i) the world through the two agents and (ii) how the agents change their states through the *g* function. Nevertheless, for the sake convenience we may assume that the world has different states for the two agents and these agents change their states through the *g* function which operates only with the answers to the questions that apply only to one of the agents.

The second world

So far we have described the first world in which the agent plays solitaire against himself and have written the program [8] which emulates the first world. The program [8] is the *g* function which describes the first world. We have also described a second world in which the agent plays against some opponent (antagonist). Now, can we also create a program which emulates the second world?

In the second world we added a statement which says "This algorithm can be executed". (This statement was to be added in the first world, because playing moves after which we are in check is not allowed in the first world, too. For the time being the program [8] allows us to make this kind of moves.) In the second world we also added the operation "Opponent executes an algorithm". In the general case that statement and that operation are unresolvable (more precisely, they are semi-resolvable).

For example, let us take the statement "This algorithm can be executed". In this particular case the question is whether the opponent can capture our king, which is fully resolvable because the chessboard is finite, has finitely many positions and all algorithms operating over the chessboard are resolvable. In the general case the algorithm may be a Turing machine and then the statement will be equivalent to a halting problem.

The same can be said of the operation "Opponent executes an algorithm". While the algorithm can be executed by many different methods, the problem of finding at least one of these methods is semi-resolvable. In the particular case of the chess game we can easily find one method of executing the algorithm, or even all methods (i.e. all possible moves). In the general case, however, the problem is semi-resolvable.

Therefore, in this particular case we can write a program which emulates the second world, provided however that we have to select the opponent's behavior because it can go in many different paths. In other words, in order to create a program which emulates the chess world, we should embed in it a program which emulates a chess player.

In the general case however, we will not be able to write a program which emulates the world we have described. Thus, the language for description of worlds is already capable of describing worlds that cannot be emulated by a computer program. We said in the very beginning that the *g* function may turn out to be non-computable. Writing a program which computes non-computable functions is certainly impossible.

However, being unable to write a program that emulates the world we have described is not a big issue, because we are not aiming to emulate the world, but write an AI program which acts on its understanding of the world (the description of the world which it has found) in order to successfully plan its future moves. Certainly, the AI program can proceed with one emulation of the world, play out some of its possible future developments and select the best development. (Essentially this is how the Min-Max algorithm of chess programs works.) I.e. making an emulation of the world would be a welcome though not mandatory achievement.

Besides being unable to produce a complete emulation of the world (when the *g* function is non-computable), AI would be unable to even figure out the current state of the world (when the possible states are continuum many). Nevertheless, AI will be able to produce a partial emulation and figure out the state of the world to some extent. For example, if there is an infinite tape in the world and this tape carries an infinite amount of information, AI will not be able to discern the current state of the world, but would be capable of describing some finite section of the tape and the information on that finite section. In other words, AI cannot find the answers to infinitely many questions, but can live with the answers to a subset of the questions.

Even the Min-Max algorithm is not a complete emulation due to combinatorial explosion. Instead, Min-Max produces partial emulation by only traversing the first few moves. If the description of the world contains a semi-resolvable rule, AI will use that rule only in one direction. An example is the rule which says that "A statement is true if there is proof for that statement". People use that rule if (i) a proof exists and (ii) they have found that proof. If a proof does not exist, the rule is not used because we cannot ascertain that there is not any proof at all.

Conclusion

We have reduced the task of creating AI to a purely logical problem. Now we have to create a language for description of worlds, which will be a logical language because it would enable the description of non-computable functions. If a language enables only the description of computable functions, it is a programing and not a logical language.

The main building blocks of our new language will be Event-Driven models. These will be the simple modules which we are going to discover one by one. With these modules we will present patterns, algorithms and phenomena.

Then we will proceed with the next abstraction and introduce objects. We will not observe the objects directly and instead will detect them by observing their properties. A property is a special

phenomenon which transpires when we observe an object which possesses that property. Thus the property is also presented through an ED model.

Our next abstraction will be the agent. Similar to objects, we will not be able to detect agents outright but will gauge them indirectly by observing their actions. The detection of agents is a difficult task. People manage to detect agents, however, they need to search them everywhere. Whenever something happens, people quickly jump into the explanation than some agent has done that. In peoples' eyes, behind every event there lurks a perpetrator which can be another human or an animal or some deity. Very seldom they would accept that the event has occurred through its own devices. AI should behave as people do and look for agents everywhere.

Once AI detects an agent it should proceed to investigate the agent and try to connect to it. To detect an agent means to conjure up an agent. When AI conjures up an existing agent, then we can say that AI has detected the agent. When AI conjures up a non-existing agent, the best we can say is that AI has conjured up a non-existing thing. It does not really matter whether agents are real or fictitious as long as the description of the world obtained through these agents is adequate and yields appropriate results.

AI will investigate agents and classify them as friends or foes. It will label them as smart or stupid and as grateful or revengeful. AI will try to connect to agents. To this end, AI must first find out what each agent is aiming at and offer that thing to the agent in exchange of getting some benefit for itself. This exchange of benefits is the implementation of a coalition strategy. Typically it is assumed that agents meet somewhere outside the world and there they negotiate their coalition strategy. But, because there is no such place outside the world, we will assume that agents communicate within the walls of the world. The principle of their communication is: "I will do something good for you and expect you to do something good for me in return". The other principle is "I will behave predictably and expect you to find out what my behavior is and start implementing a coalition strategy (engage in behavior which is beneficial to both of us)".

This how we communicate with dogs. We give a bone to a dog and right away we make friends. What do we get in return? They will not bite us or bark at us, which is a fair deal. As time goes by the communication may become more sophisticated. We may show an algorithm to the agent and ask him to replicate it. We can teach the dog to "shake hands" with a paw. Further on, we can get to linguistic communication by associating objects with phenomena. For example, a spoken word can be a phenomenon and if this phenomenon is associated with a certain object or algorithm, the agent will execute the algorithm as soon as he hears the word. E.g. the dog will come to us as soon as it hears its name or bring our sleepers when it hears us saying "sleepers".

You see that through its simple constituent modules, the language for description of worlds can describe quite complicated worlds with multiple agents and complex relationships among the agents. The superstructure we build on these modules cannot hover in thin air and should rest on some steady fundament. Event-Driven models are exactly the fundament of the language for description of worlds and the base on which we will develop all abstractions of higher order.

References

[1] Yiannis N. Moschovakis (2001). What is an algorithm? *Mathematics unlimited – 2001 and beyond, edited by B. Engquist and W. Schmid, Springer, 2001, pages 919-936.*

[2] Yiannis N. Moschovakis (2018). Abstract recursion and intrinsic complexity. *Cambridge University Press, Lecture Notes in Logic, Volume 48, ISBN: 9781108415583.*

[3] Xi-Ren Cao (2005). Basic Ideas for Event-Based Optimization of Markov Systems. *Discrete Event Dynamic Systems: Theory and Applications, 15, 169–197, 2005.*

[4] Xi-Ren Cao, Junyu Zhang (2008). Event-Based Optimization of Markov Systems. *IEEE TRANSACTIONS ON AUTOMATIC CONTROL, VOL. 53, NO. 4, MAY 2008.*

[5] Dimiter Dobrev (2013). Giving the AI definition a form suitable for the engineer. *arXiv:1312.5713 [cs.AI].*

[6] Dimiter Dobrev (2018). Event-Driven Models. *International Journal "Information Models and Analyses", Volume 8, Number 1, 2019, pp. 23-58.*

[7] Dimiter Dobrev (2019). Before we can find a model, we must forget about perfection. *arXiv:1912.04964 [cs.AI].*

[8] Dimiter Dobrev (2020). AI Unravels Chess. *http://dobrev.com/software/AI_unravels_chess.zip*.

[9] Dimiter Dobrev (2020). Strawberry Prolog, version 5.1. *http://dobrev.com/* .