Serdica Journal of Computing 17(1), 2023, pp. 17-54, 10.55630/sjc.2023.17.17-54
Published by Institute of Mathematics and Informatics, Bulgarian Academy of Sciences

Language for Description of Worlds. Part 2: The Sample World

Dimiter Dobrev

Institute of Mathematics and Informatics,
Bulgarian Academy of Sciences, Bulgaria
d@dobrev.com

Abstract

This is the second part of the paper. In this part we will use the world of the chess game in order to create the language we are looking for. We will show how a complex world can be described in a simple and understandable way. Before describing the movement of chess pieces, we will need to extend the concept of *algorithm*. The new concept describes the algorithm as a sequence of actions performed in an arbitrary world. In the meaning of the new concept, a cooking recipe is also an algorithm. If we look at a world in which there is an infinite tape and a head which travels over the tape, then the algorithm of that world will be a Turing machine. This means that the new concept of algorithm is a generalization of the old one. Computer programs are algorithms both in the new concept and in the old one, however, there are many other sequences of actions which extend the concept.

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

ACM Computing Classification System 2012: Computing methodologies → Artificial intelligence → Philosophical/theoretical foundations of artificial intelligence

Mathematics Subject Classification 2020: 68T01

Received: April 11, 2023, *Accepted:* June 20, 2023, *Published:* July 7, 2023

Citation: Dimiter Dobrev, Language for Description of Worlds. Part 2: The Sample World, Serdica Journal of Computing 17(1), 2023, pp. 17-54,
https://doi.org/10.55630/sjc.2023.17.17-54

1 Introduction

This is the second part of the paper. In Part 1 [1] we already introduced the basic concepts which we will need going forward. In this second part, we will create the language we are looking for by using the description of a concrete world. We will describe the world of the chess game. The chess game will be presented in two versions: a game with a single player (i.e. a game in which we play against ourselves), and a game between two players.

By using Event-Driven (ED) models, we will generalize the concept of *algorithm* and will apply the newly-obtained concept in order to describe the algorithms which define the movement of the various chess pieces. We will show that Turing machines are a special case of the newly-obtained *algorithm* concept. For this purpose we will use a third world in which we have an infinite tape and head which travels over the tape.

Contributions

1. A definition of the concept *algorithm*. We have presented the algorithm as a sequence of events in an arbitrary world. Further on, we present the Turing machine as an ED model found in a special world where an infinite tape exists. Thus we prove that the new definition generalizes the *Turing machine* concept and expands the *algorithm* concept.

2. A language for description of worlds such that the description can be searched automatically without human intervention.

How is this paper organized. First (in Section 2) we will identify the particular world which we are going to describe. Then (in Section 3) we will prove that the known tools for description of worlds are not appropriate for the world in question.

In Section 4 we will describe some simple patterns and will present them by using ED models (such as the patterns Horizontal and Vertical).

In Section 5 we will define the rules to which the chess pieces move. For this purpose we will need to expand the *algorithm* concept. The algorithms to which chess pieces move will also be presented through ED models.

In Section 6 we will present the chess pieces as objects. The objects will be an abstraction of higher order. They will be defined through properties. The property is something concrete and it will be defined through an ED model.

In Section 7 we will add a second player. For this purpose we will make another higher-order abstraction. This will be the abstraction *agent*. We will not observe agents directly and instead will gauge them through their actions.

Agents will make the world non-computable, but if we add some non-computable rule then the world can become non-computable even without agents.

Finally, in Section 8 we will look at the agents and various interplays between them.

2 The chess game

Which concrete world are we going to use in order to create the new language for description of worlds? This will be the world of chess.

Let us first note that we will want the world to be partially observable because if the agent can see everything the world will not be interesting. If the agent sees everything, she will not need any imagination. The most important trait of the agent is the ability to imagine the part of the world she does not see at the current moment.

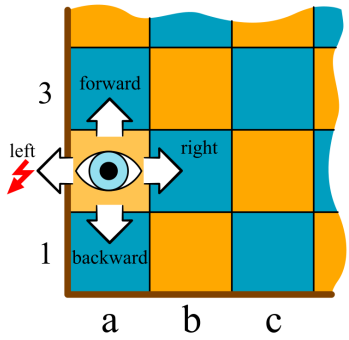

Figure 1: Position of the eye

For the world to be partially observable we will assume that the agent sees just one square of the chessboard rather than the entire board (Figure 1). The agent's eye will be positioned in the square she can see at the moment, and the agent will be able to move that eye from one square to another so as to monitor the whole board. Formally speaking, there is not any difference between seeing the whole board at once and exploring it by checking one square at a time – in either case one gets the full picture. There will not be any difference only if you know that by moving your sight from one square to another you will monitor

the whole chessboard. In practice the agent does not know anything, so she will need to conjure up the whole board, which however will not be an easy process and will require some degree of imagination.

In Figure 1, the agent has her eye in square **a2** and can move it in all the four directions. (Right now she cannot move it left because this is the edge of the board and a move to the left would be incorrect.) In addition to moving the eye in the four directions, the agent can perform two other actions: *"Lift the piece you see right now"* and *"Put the piece you lifted in the square you see right now"*. We will designate the two additional actions as *Up* and *Down*. These six actions will enable the agent monitor the chessboard and move the chess pieces, and that's everything one needs to play chess.

2.1 A chess game with a single player

We will examine two versions of a chess game – a game with a single player and a game with two players.

How do you play chess with a single player? You first move some white piece, then turn the board around, play some black piece and so forth.

We will start by describing the more simple version in which the agent plays against herself. This version is simpler because in that world there is only one agent and that agent is the protagonist. Next we will examine the more complicated version wherein there is a second agent in the world and that second agent is an opponent of the protagonist.

The question is what can be the goal when we play against ourselves?

2.2 The goal

While the authors of most papers dedicated to AI choose a goal, in this paper we will not set a particular goal. All we want is to describe the world, and when we figure out how the world works we will be able to set various goals. In chess for example our goal can be to win the game or lose it. When we play against ourselves, the goal can vary. When we play from the side of the white pieces our goal can be *"white to win"* and vice versa.

Understanding the world does not hinge on the setting of a particular goal. The Natural Intelligence (the human being) usually does not have a clearly defined goal, but that does not prevent human beings from living.

There are two questions: "What's going on?" and "What should I do?" Most authors of AI papers rush to answer the second question before they have answered the first one. In other words, they are looking for some policy, and a

policy can only exist when there is a goal to be pursued by that policy. We will try to answer only the first question and will not deal with the second one at all. Hence, for the purposes of the present paper we will not need a goal.

In most papers the goal is defined through *rewards* (the goal is to collect as many rewards as possible). When referring to Markov decision process (MDP) we will assume that rewards have been deleted from the definition because we need them only when we intend to look for a policy, while this paper is not about finding policies.

3 Related work

Can the chess world be described using the already known tools for description of worlds? We will review the tools known at this time and will prove that they are not appropriate.

3.1 Markov decision process

The most widely used tool for description of worlds is Markov decision process (MDP). Can we use MDP for describing the kind of world selected in this paper? Let us first note that we will have to use Partially observable MDP (POMDP) because the world we are aiming to describe is Partially observable.

Certainly, the chess world can be presented as a POMDP, but how many states will this take? We will need as many states as the positions on the chessboard are, which is an awful lot (in the range of 1045 according to [2]). We will even need some more states because in addition to the chessboard position the state must remember the whereabouts of the eye. This means 64 times more states, which is a little more because adding two zeroes to a large number seems an insignificant increase. Thus, we do not perceive the numbers 1045 and 1047 as much different.

If we wish to describe this kind of POMDP in tabular form, the description will be so huge that storing it would be beyond the capacity of any computer memory. Of course storing the description is the least problem. A much more serious issue is that we should find and build that table on the basis of our lived experience which means that for such a huge table we would need enormously huge lived experience (almost infinite).

Therefore, efforts to find a POMDP description of the chess are bound to fail.

3.2 Situation calculus

The first language for description of worlds was proposed by Raymond Reiter and this is *Situation Calculus* described in [3].

The chess world can indeed be presented through the formalism proposed by Raymond Reiter, but we will run into two problems (a small one and big one).

The first (smaller) problem is that in order to present the state of the world obtained after a certain action, Reiter uses a functional symbol. Thus, Reiter assumes that the next state of the world is unambiguously defined. That assumption would not be a problem for deterministic worlds such as the chess game, but would be an issue when it comes to a dice game. The problem of course is not a big one because we can always assume that the next state is determinate even if we do not know which one it is. (In other words, we can assume that there is some destiny which defines the future in an unambiguous manner, although it does not help us predict what is going to happen.)

In Section 3.2. of their publication of 2001, Boutilier, Reiter and Price [4] attempt to resolve the first (smaller) problem by replacing each step with two steps (plies). Instead of a single step of the agent they suggest two plies – *agent ply* and *nature ply*. The idea is that the agent step is non-deterministic because the world (nature) can respond in a variety of ways, so if we decouple the agent's action from nature's response then the output from each ply would be deterministic. Essentially we use the same idea by creating a Simple MDP (further down in this paper).

The second (big) issue with Situation Calculus is Reiter's implied assumption that there is some human being (programmer) who has figured out how the world is designed and will describe that design using first order formulas. All of us wish to get to a description of the world by first order formulas, but the objective is to ensure that the description can be found automatically, i.e. without human intervention. While this paper actually provides a manmade description of the chess game, our objective is to come up with a machine-searchable description and the one provided here can indeed be searched and found automatically.

4 Description of the chess game

4.1 Computer emulation

We have emulated the chess-game world by the computer program [5] written in the language Prolog [6]. The rules of the game used by that program are presented as ED models.

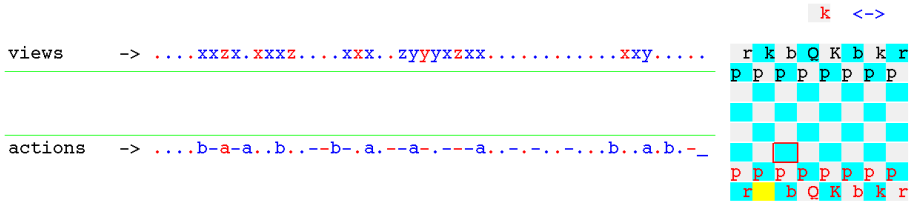

Figure 2: Chess game computer emulation

When you start the program [5], in the bottom of the screen you will see the visuals provided in Figure 2.

The left-hand side of Figure 2 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.

All the agent can see is the left-hand side of Figure 2. 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).

4.2 We use coding

The agent will be able to do 8 things: move her gaze (the square currently observed) in the four directions, lift the piece she sees at the moment and drop the lifted piece in the square she 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 [7] that we should avoid excessive coding because the world is complicated enough and we do not want to complicate it further. However, the coding here is not excessive because it replaces eight actions with four and therefore simplifies the world rather than complicate it.

4.3 Two void actions

Why do we introduce two actions that mean "I do nothing"? Actually, when the agent just stays and does nothing, she observes the world. The question is, will she be a passive observer or will she 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 her to detect the pattern (property), especially since she 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 she observes will be much clearer and more quickly detectable.

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

4.4 One, two, three

The first pattern which will exist in this particular world (the game of chess) stems from our division of the steps in three groups. Let us name this pattern "One, two, three". The pattern is modeled in Figure 3.

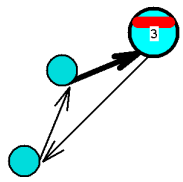

Figure 3: One, two, three

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

The pattern is presented through an Event-Driven (ED) model. 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").

4.5 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 [8].

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 "One, two, three" 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 just a regular 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 relatively permanent or transient phenomena (a relatively permanent phenomenon is one which always occurs in a given state). 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.

4.6 Horizontal and vertical

The next Event-Driven model we need for our description of the world is the Horizontal model (Figure 4).

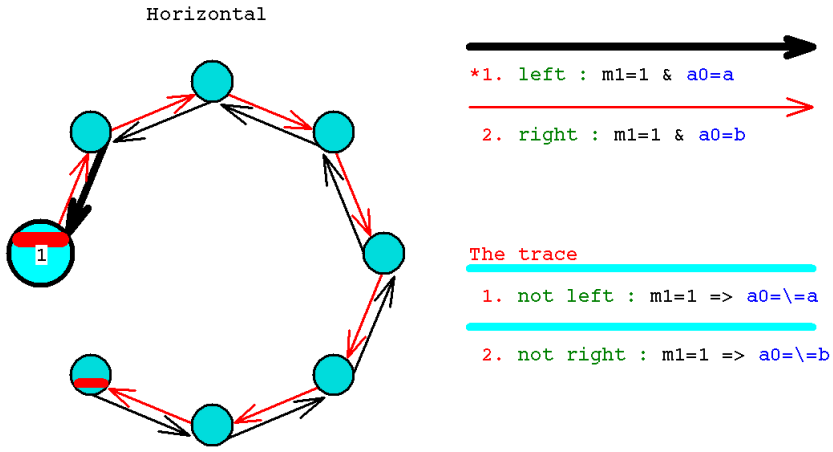

Figure 4: Horizontal pattern

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 her 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 8 strides wide, you will find that after making 7

strides you cannot continue in the same direction. 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 $left$.

The next model is shown on Figure 5. This model is very similar to the previous one.

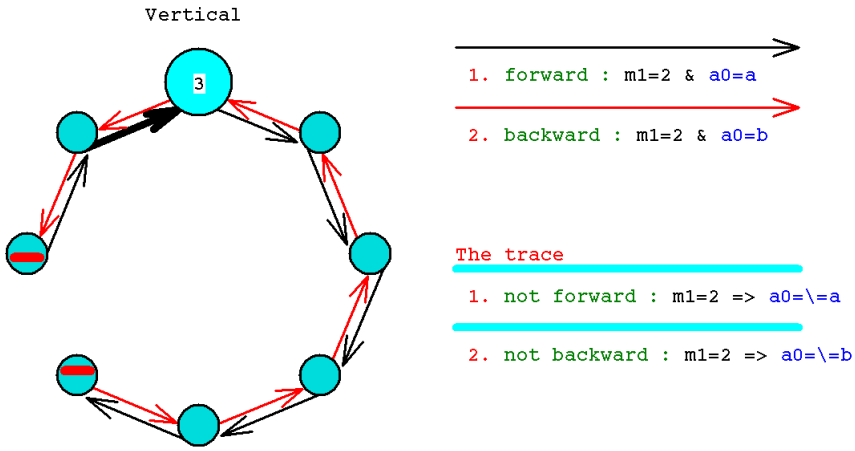

Figure 5: Vertical pattern

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.

4.7 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, for each state we included an indication showing 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 in the current moment. 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:

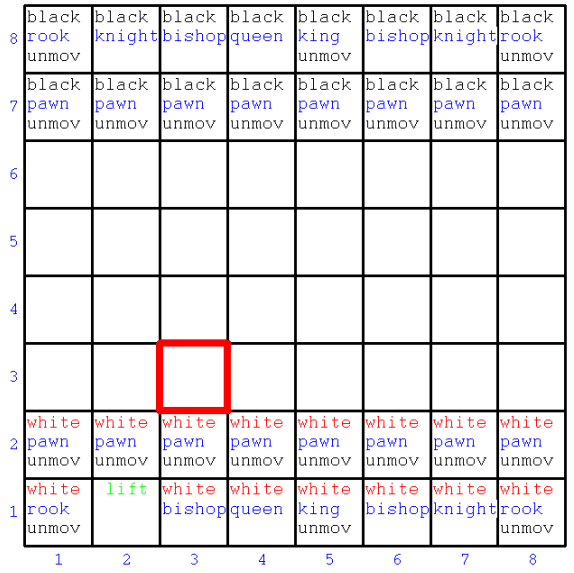

Figure 6: The chessboard as a moving trace

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 7). 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 a piece is impossible* and in state 1 *dropping a 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 current 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 what we have dropped, in this case a white knight.)

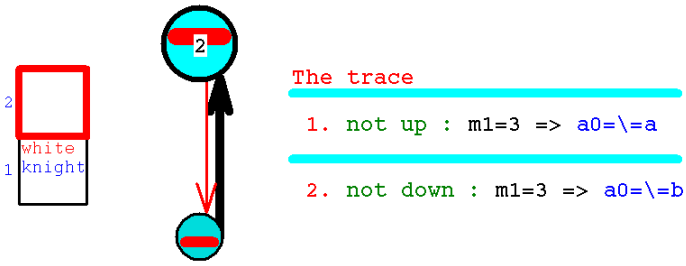

Figure 7: Which is the lifted piece?

We said that the language for description of worlds will tell us which the current state of the world is. Where is this state stored? At two locations – first, the current state of each ED model and second, the moving traces. For example, in Figure 6 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 can be a *belief* if different states have different probabilities). 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.

5 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.

Most papers do not distinguish between an algorithm and a computable function. This is not correct because the algorithm is a sequence of actions while the computable function is the result from the execution of that sequence of actions. We should differentiate sequences of actions from results. E.g. a pancake recipe is not the same as a real pancake. The result from the execution of an algorithm depends on the specific world in which we execute that algorithm. This means that in a different world the pancake making algorithm may produce a different result. That result can be, for example, a computable function or a spacecraft.

5.1 What is an algorithm

For most people an algorithm is a Turing machine. The reason is that they only look at $\mathbb{N} \to \mathbb{N}$ 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 consider 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 a directed graph with multiple branches and not as a road without any furcations.

5.2 The formal definition

Our definition of algorithms will be very similar to that of phenomena except that we will add a result to the terminal conditions.

Definition 30. An algorithm will be the 3-tuple $\langle Triggers, Model, Conditions\rangle$. Here $Triggers$ is a set of trigger points, $Model$ is an ED model and $Conditions$ is a set of terminal conditions, including some results.

Definition 31. A trigger point will be the 3-tuple $\langle e, [a, b], s\rangle$. Here e is an event, $[a, b]$ is some probability (probability interval) and s is the state of the ED model from which the algorithm sets off.

Definition 32. A terminal condition with a result will be the 3-tuple $\langle e, [a, b], r\rangle$. Here e is an event, $[a, b]$ is some probability (probability interval)

and r is an element of some set *Results*. The set *Results* is finite or countable, at most.

Our typical understanding of a phenomenon is something which happens on its own devices, while an algorithm is perceived as something which is executed by someone else. Our definition here does not differentiate the two concepts in this way. The only difference in our case is the assumption that an algorithm can return some result (such as *yes* or *no*).

Definition 33. The execution of an algorithm is a segment of life which starts with a trigger point and ends in a terminal condition with a result. Also, the ED model pattern has to be fulfilled within that segment.

When dealing with a Railway track type of algorithm we assume that there is some *trace* which prevents us from exiting the execution of the algorithm before it comes to its end (before the end of the rail track). That is, the *trace* will indicate that such a departure from the algorithm is impossible. When the algorithm at hand is of Mountain footpath type, we assume that we have a *trace* indicating that any termination before the end of the execution will lead to the occurrence of some additional events (e.g. falling in an abyss). In this case, we have to add one more terminal condition (the event will be "exit the algorithm before the end of execution", the probability of which will be 1 and the result from which will be *algorithm is interrupted*). When the algorithm is of Going home type, the assumption is that there is no *trace* which prevents us from exiting before the end of the algorithm, nor a trace which indicates that some additional events will occur upon exit. However, we will need to add an additional terminal condition the result from which is *algorithm is interrupted*. This additional condition will allow the termination of the algorithm before its execution has come to an end. That is, we will still have some execution of the algorithm, but that execution will end with *algorithm is interrupted*.

5.3 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 put her in some rail track, she will struggle a lot in order to make a correct move. Moreover, we should consider how the agent would understand the world. How would she discover these algorithms? If we put her on a rail track, she will learn the algorithm – like it or not – but if we let her loose she would have hard time trying to guess what the rules for movement 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 the algorithm more readily if you demonstrate hands-on how it works.

What will be the gist of our algorithms? These will be Event-Driven models. There will be some event which will be the trigger point of the algorithm which triggers its execution (an event which sets off from state 1 with probability of one) and another terminating condition which will put an end to the execution of the algorithm. Later on we will clone the terminating condition in two (*real move* and *fake move*). Each chess piece will have its algorithm.

5.4 The king and knight algorithms

The king's algorithm (Figure 8) will be the simplest one. The trigger event will be *king lifted*. The initial state will be state 1 (this applies to all algorithms described here). The events will be four ($left$, $right$, $forward$ and $backward$).

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 8, 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

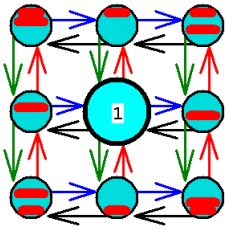

Figure 8: The algorithm of the king

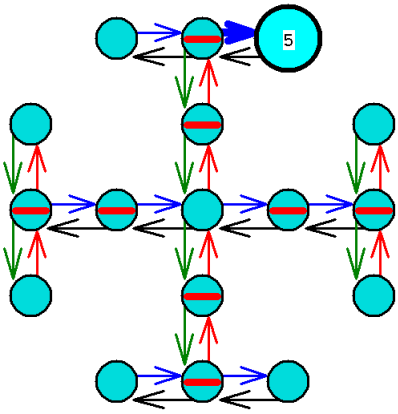

Figure 9: The algorithm of the knight

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 9). 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 9, 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.

5.5 The rook and bishop algorithms

Although with less states, the rook's algorithm is more complex (Figure 10). 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.

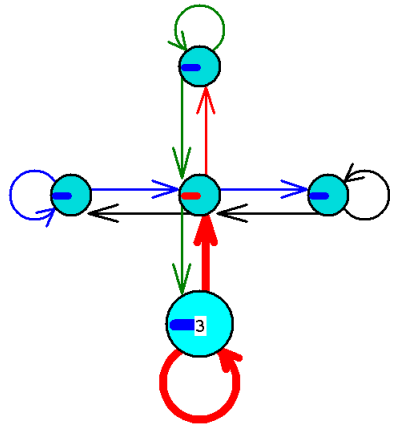

Figure 10: The algorithm of the rook

Even more complicated is the bishop's algorithm (Figure 11). 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

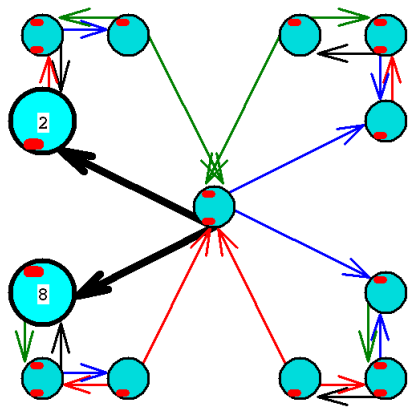

Figure 11: The algorithm of the bishop

will be resolved. (In Figure 11 we have marked only the *no-forward* and the *no-backward traces*.)

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.

5.6 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 4. 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 for description of the world, 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. 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 usually use Turing machines in order to compute $\mathbb{N} \to \mathbb{N}$ 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 a countable number of configurations, however, all possible configurations of the tape 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 letters (0 and 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:

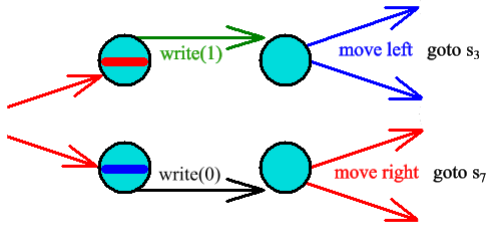

Figure 12: Each command replaced with four states

In Figure 12, 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 12 and then the individual quaternaries are interconnected. For example, the quaternary in Figure 12 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 Going 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 she 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 her action. The agent can influence even those events which are not driven by her actions. This is described in [9]. For these events the agent may have some "preference" and by her "preferences" the agent could have some influence on whether an event will or will not occur.

5.7 Related work

Importantly, this paper 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 made by Moschovakis [10,11]. 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 [10]. 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 [10] 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."

6 Objects

6.1 Properties

Having defined the term *algorithm*, we will try to define another fundamental concept: *property*. For the definition of this concept 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. Phenomena 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).

6.2 What is an object?

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 dispense of objects and simply imagine that some properties come and go, i.e. some phenomena appear and disappear. However, the abstraction *object* is mandatory for understanding complex worlds.

6.3 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, she may spot various patterns. The presence or absence of each of these patterns will be the information which the agent will receive for the square she 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 13 depicts the property *king*. 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 king would look like. How the king looks like does not matter. What matters is that the king should have a certain appearance such that it can be recognized by the agent. In other words, the king should have some face, but how that face would look like is irrelevant.

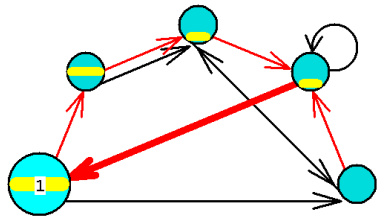

Figure 13: How does the king look like?

In our program [5] 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 contra-

dictory. 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.

7 A two-players chess game

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, her determinacy would be too complicated for us to describe it.

7.1 A deterministic world

So far we have described a chess world where the agent plays solitaire against herself. We have written a program [5] which contains a simple description of this world and by this description emulates the world. Run this program and see how simple that description happens to be. The description consists of 24 modules presented in the form of Event-Driven models (these models are directed graphs with a dozen states each). The ED models of the chess game belong to three types: (5 patterns + 9 algorithms + 10 properties = 24 models). In addition to the ED models there are also two moving traces (i.e. two arrays). We added also six simple rules which we need as well. 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. This rule looks like this:

$$up, here \Rightarrow copy(Lifted) \tag{1}$$

(If we lift a piece and if we are in the square $\langle X, Y \rangle$, then the property $Lifted$ will replace all properties which are in this square at the moment.)

We are able to formulate these rules owing to the fact that we have the context of the chessboard (the moving trace from Figure 6). 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. In the demonstration program [5] the agent plays randomly. Of course the agent's actions are not important. What matters is the world and that we have described it.

The description thus obtained is deterministic, i.e. the initial state is determined and every next state is determined. 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 we 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 model may be non-deterministic. For most authors, non-deterministic implies that for each possible event there is one precisely defined probability. In [8,9] 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.

7.2 Impossible events

We said the world would be more interesting if we do not play 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*. Previously we defined that event 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*). This change produces a world where the agent cannot switch sides.

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 do 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 14 depicts the algorithm which describes how we switch sides (turn the chessboard around) and capture the king. If an execution of that algorithm exists, then the move is incorrect. (Even if an execution exists the algorithm cannot be executed, because it includes an impossible event.)

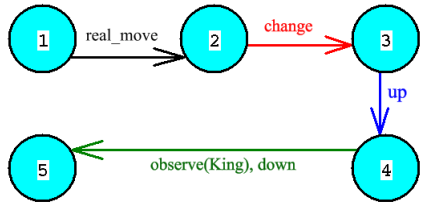

Figure 14: How do we switch sides?

This algorithm is more consistent with our understanding of algorithms. While the algorithms of the chess pieces were directed 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 14. 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) $\Rightarrow$ 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 paper, 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).

7.3 The second agent

The algorithm in Figure 14 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 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 15.

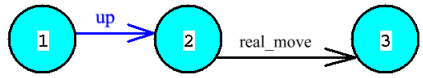

Figure 15: Algorithm of the black-pieces agent

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 would take 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 she assist us or try to disrupt

us). The agent's smartness is also important (because she 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, she 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.

7.4 Agent-specific state

We assumed here that the second agent sees a distinct state of the world. I.e. the second agent has her distinct position $\langle x, y \rangle$ on the chessboard (the square which she 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 according to the same model but the memory of that model (the state of the world) is specific to each agent. 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 6 is the same for both). We will further assume that each agent has her own coordinates and a specific color for her 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 7, 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 15 would become heavily complicated. The antagonist would first turn the chessboard around (*change*), then play her 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 her starting coordinates $\langle x, y \rangle$ (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 do with his.

Note: It is not very accurate to say that the world has distinct states for both 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 complicated states. Let the new set of states be S''. We can assume that $S'' = S \times S$. 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?".

Thus, from the model where the states are S we have derived a new model where the states are S''. The difference between S and S'' is that the states in S describe the state of one agent (without telling us which is that agent), while the states in S'' describe the states of the two agents. (In both cases the overall state of the world is described as well.) The new model describes (i) the world through the two agents and (ii) how the agents change their states according to the first model. Nevertheless, it is more natural to assume that the world has different states for the two agents and these agents change their states according to the first model which operates only with the questions that apply only to one of the agents.

7.5 Non-computable rule

So far we have described the first world in which the agent plays solitaire against herself and have written the program [5] which emulates the first world. The program [5] is a model 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 [5] 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 undecidable (more precisely, they are semi-decidable).

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 decidable because the chessboard is finite, has finitely many positions and all algorithms operating over the chessboard are decidable. 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-decidable. 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-decidable.

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 model may turn out to be non-computable. Writing a program which computes non-computable model 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 model 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.

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-decidable 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.

8 Agents

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.

8.1 Interaction between agents

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 policy. Typically it is assumed that agents meet somewhere outside the world and there they negotiate their coalition policy. 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 policy (engage in behavior which is beneficial to both of us)".

This is 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 her 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 is a phenomenon and if this phenomenon is associated with a certain object or algorithm, the agent will execute the algorithm as soon as she 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".

8.2 Signals between agents

When it comes to interaction or negotiation, we need some sort of communication. This takes us to the signals which agents send to their peers. We do not mean pre-arranged signals, but ones which an agent chooses to send and the others decipher on the basis of their observations. One example is "Pavlov's Dog" [12]. Pavlov is the agent who decides to send a signal by ringing a bell before he feeds the dog. The other agent is the dog which manages to comprehend the signal.

Where an agent sends a signal to another agent, the latter need not necessarily realize that this is a signal and has been send by someone else who is trying to tell her something. In the previous example, Pavlov's dog has not any idea that Pavlov is the one ringing the bell to signal that lunch is ready. The dog simply associates the *ringing* event to the $feeding$ event. Thus, when we send a signal we can remain anonymous. This ultimately means that we can influence another agent without that agent ever realizing that she is under somebody else's influence.

Another way of sending a signal is to show something (provide some information). For this purpose we need to know what the other agent can see and when. For example, when a dog growls at us, it shows us its fangs. We see that the dog has fangs – something which we know by default – but what we realize in this case is that the dog has decided to remind us of this fact and understand the message as "The dog issues a warning that it may use its fangs against us".

In addition to natural signals (ones which we can guess ourselves), there may be pre-established signals. Let us have a group of agents who have already established some signals between them. When a new agent appears, she may learn a signal from one of the agents and then use it in her communication with other agents. An example for such signals are the words in our natural language.

We learn the words from an agent (e.g. from our mothers) and then use the same words in order to communicate with other agents.

8.3 Exchange of information

When agents communicate, they can exchange information in order to coordinate their actions or to negotiate. An example of information exchange is when an agent shares some algorithm with another agent. The algorithm can be described in a natural language, i.e. it can be presented as a sequence of signals (words) where each signal is associated with an object, phenomenon or algorithm. For example, when we tell somebody how to get to a shop, we explain this algorithm by using words. When we say "Open the door" we rely that the other agent will associate the word "door" with the object *door* and the word "open" with the algorithm *open*. In other words, we rely that the agent knows these words and has an idea of the objects associated with these words.

If we assume that the agent keeps the algorithms in her memory in the form of Event-Driven models, then the agent should be able to construct an ED model from a description expressed in a natural language and vice versa – describe some ED model in natural language (as long as the agent knows the necessary words).

8.4 Communication interface

When creating the AI's world, we need to equip it with some communication interface to enable it communicate with other agents.

For example, when building a self-driving vehicle, we must give that vehicle some face so that it can communicate with pedestrians and other drivers. Indeed, vehicles have horns and blinkers, but this is not sufficient for full communication. It would be a good idea to add some screen which expresses various emotions. Smiling and winking will be very useful functions.

We usually try see where the other driver is looking at because it is very import for us to know that the other driver has seen us. If this face (screen) is able to turn to our side, it would be an indication that we have been seen.

8.5 Related work

Many authors deal with the interaction between agents. Their papers however do not tell us how AI will discover the agents as they assume that the agents have already been discovered and all we need to do is set out rules for

reasonable interaction. Goranko, Kuusisto and Rönnholm [13] for example look at the case where all agents are friends and their smartness is unlimited. Agents in [13] communicate on the basis that they can foretell what the others would do (relying on the assumption that all agents are friends and are smart enough to figure out what would be beneficial for everybody). The most interesting aspect in [13] is that the paper raises the issue about the hierarchy between agents (who is more important) and about their hastiness (how patient is each agent). These are principles which real people apply in the real world and it therefore makes sense for AI to also use these principles.

Interaction between agents is as complicated as interaction between humans. As an example, agents in Mell, Lucas, Mozgai and Gratch [14] negotiate and can even cheat each other.

Gurov, Goranko and Lundberg [15] as well as the present paper deal with a multi-agent system where the agents do not see everything (Partial Observability). The main difference between [15] and this paper is that in [15] the world is given (is described by one relation) while in this paper the world is not given and is exactly the thing that has to be found.

9 Future work

In this paper we provided a manual description of a world (the chess game) and created the computer program [5] which emulates the world on the basis of the manual description. Our next problem is the inverse one, namely create a program which should automatically arrive at the same description of the world as the one which we have described manually. That program will use the emulation of the world [5], thanks to it will "live" inside the world and will have to understand it (i.e. to describe it).

In this case we might be tempted to play with marked cards because the program we make has to find a thing, and we know in advance what this thing is. Of course we should not yield to this temptation because we will end up with a program which is able to understand only and exclusively the particular world. It would be much better if our program is able to understand (describe) any world. This however is a tall order because essentially it asks us to build AI. Accordingly, instead of aiming at a program which is capable to understand any world, we would be happy with a program which can understand the given world [5] and the worlds that are proximal to it. The larger the class of worlds our program is able to understand, the smarter that program will be.

10 Conclusion

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.

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 are Event-Driven models. These are the simple modules which we are going to discover one by one. With these modules we will present patterns, algorithms and phenomena.

We introduced some abstractions. Our first abstraction were objects. We cannot observe the objects directly and instead gauge 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.

Then we introduced another abstraction – agents. Similar to objects, we cannot observe agents directly and can only gauge them through their actions.

We created a language for description of worlds. This is not the ultimate language, but only a first version which needs further development. We did not provide a formal description of our language and instead exemplified it by three use cases. That is, instead of describing the language we found the descriptions of three concrete worlds – two versions of the chess game (with one and two agents, respectively) and a world which presents the functioning of the Turing machine.

Note: It is not much of a problem to provide a formal description of a language which covers all the three worlds, but we are aiming elsewhere. The aim is to create a language which can describe any world, and provide a formal description of that language. This is a more difficult problem which we are yet to solve.

We demonstrated 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] D. Dobrev, Language for Description of Worlds. Part 1: Theoretical Foundation, *Serdica Journal of Computing*, 16(2):101-150, 2022.

[2] J. Tromp, Chess Position Ranking, *John Tromp's personal web page*, 2021.

[3] R. Reiter, *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*, MIT Press, Cambridge, 2001.

[4] C. Boutilier, R. Reiter, B. Price, Symbolic Dynamic Programming for First-Order MDPs, *Proceedings of the International Joint Conference of Artificial Intelligence*, 1:690–700, 2001.

[5] D. Dobrev, AI Unravels Chess, *program published on dobrev.com*, 2020.

[6] D. Dobrev, Strawberry Prolog, ver. 5.1, *program published on dobrev.com*, 2020.

[7] D. Dobrev, Giving the AI definition a form suitable for the engineer, *arXiv: 1312.5713 [cs.AI]*, 2013.

[8] D. Dobrev, Event-Driven Models, *International Journal "Information Models and Analyses"*, 8(1):23–58, 2019.

[9] D. Dobrev, Before We Can Find a Model, We Must Forget about Perfection, *Serdica Journal of Computing*, 15(2):85–128, 2021.

[10] Y. Moschovakis, What is an algorithm?, *Mathematics Unlimited – 2001 and Beyond*, edited by B. Engquist and W. Schmid, Springer, 2001.

[11] Y. Moschovakis, Abstract recursion and intrinsic complexity, *Lecture Notes in Logic*, 48, Cambridge University Press, 2018.

[12] I. Pavlov, The work of the digestive glands, *Charles Griffin & Company, Limited*, London, 1902.

[13] V. Goranko, A. Kuusisto, R. Rönnholm, Gradual guaranteed coordination in repeated win-lose coordination games, *24th European Conference on Artificial Intelligence – ECAI 2020, Santiago de Compostela, Spain*, Frontiers in Artificial Intelligence and Applications, 325:115–122, 2020.

[14] J. Mell, G. Lucas, S. Mozgai, J. Gratch, The Effects of Experience on Deception in Human-Agent Negotiation, *Journal of Artificial Intelligence Research*, 68:633–660, 2020.

[15] D. Gurov, V. Goranko, E. Lundberg, Knowledge-based strategies for multi-agent teams playing against Nature, *Artificial Intelligence*, 309:103728, 2022.