A (hopefully) Unbiased Universal Environment Class for Measuring Intelligence of Biological and Artificial Systems (EXTENDED VERSION)

José Hernández-Orallo

DSIC, Univ. Politècnica de València, Camí de Vera s/n, 46020 Valencia, Spain.
jorallo@dsic.upv.es

Abstract. The measurement of intelligence is usually associated with the performance over a selection of tasks or environments. The most general approach in this line is called Universal Intelligence, which assigns a probability to each possible environment according to several constructs derived from Kolmogorov complexity. In this context, new testing paradigms are being defined in order to devise intelligence tests which are anytime and universal: valid for both artificial intelligent systems and biological systems, of any intelligence degree and of any speed. And these tests should be adaptive and feasibly administered in a finite (short) period of time. In this paper, we address one of the pieces in this puzzle: the definition of a general, unbiased, universal class of environments such that they are appropriate for intelligence tests. By appropriate we mean that the environments are discriminative (sensitive to the examinee's actions and unbiased for random agents) and that they can be feasibly built, in such a way that the environments can be automatically generated and their complexity can be computed. The environment class we present in this paper fulfils these properties and it can be used to construct intelligence tests and, in particular, the first anytime universal intelligence test.

Keywords: Intelligence measurement, artificial intelligence, Kolgomorov complexity, Game generators, Turing-complete environments, artificial life, multi-agent systems.

1 Introduction

This paper presents a feasible environment class which can be used to test intelligence of humans, non-human animals and machines. The environment class is developed under the theory presented in [6], which is, in turn, based on [20][8][3]. Following a series of works in the late 1990s using Kolmogorov complexity, compression, Solomonoff's prediction theory and MML inductive inference, etc., which developed or extended previous tests and definitions of intelligence (see, e.g, [3], [2] [7], [10], [8], [9] [11]), the theory in [6] presents the first general and

feasible intelligence test framework, which should be valid for both artificial intelligent systems and biological systems, of any intelligence degree and of any speed. The test is not anthropomorphic, is gradual, is anytime and it is exclusively based on computational notions, such as Kolmogorov complexity. And it is also meaningful, since it averages the capability of succeeding in different environments. The key idea is to order all the possible action-reward-observation environments by their Kolmogorov complexity and to use this ordering to make a sample. In order to make this feasible (in contrast to [20]), in [6] several constraints are imposed on the environments, such as considering time and using a time-bounded and computable version of Kolmogorov complexity (see e.g. [22]), and also making rewards balanced and that the environments are sensitive to the agent actions. The environments can then be used to construct adaptive (anytime) tests in order to evaluate the intelligence of any kind of agent. The test configures a new paradigm for intelligence measurement which dramatically differs from the current specific-task-oriented and ad-hoc measurement used both in artificial intelligence and psychometrics.

The previous theory, however, does not make the choice for *an* environment class, but just sets some constraints on the kind of environments that can be used. Consequently, one major open problem is to make this choice, i.e., to find a proper (unbiased) environment class which follows the constraints and, more difficult, which can be feasibly implemented. Once this environment class is identified, we can use it to generate environments to run any of the tests variants introduced in [6]. Additionally, it is not only necessary to determine the environment class, but also to determine the universal machine we will use to determine the Kolmogorov complexity of each environment, since the tests only use a (small) sample of environments, and the sample probability is defined in terms of the complexity.

We know that for universal (i.e. Turing-complete) machines, the Kolmogorov complexity of an environment computed with a reference universal machine differs only by a constant from the Kolmogorov complexity of the environment computed with a different reference universal machine, i.e. $\forall x : |K_U(x) - K_V(x)| \leq c(U,V)$. The value of $c(U,V)$, which depends on U and V, is relatively small for sufficiently long x (see, e.g., [22] for a formal proof). Similar things happen with variants of Kolmogorov Complexity, such as the time-bounded variant suggested in [6] to calculate the complexity of an environment. Although this constant is small for many strings (even for many short strings) we can only affirm this in general for long strings. Typically, because of time limitations and the intelligence levels of the subjects we will test, we will be working with small environments[1]. Consequently, the constant (especially for short tests) is important, since using

[1] [14] suggests to only include environments of complexity greater than a positive integer L. This reduces the dependency on the reference machine. However, the exclusion of simple environments is a problem for evaluating subjects with low intelligence levels. Additionally, large environments usually require more time for an agent interacting inside in order to get a reliable assessment.

a specific universal machine could, in the end, constitute a strong bias for some subjects. The idea is then to choose a mostly unbiased reference machine.

However, the reference machine for environments is necessarily an arbitrary choice. When we say necessary, we mean that it is necessary in general, not only in our framework, because of the impossibility of finding a canonical reference machine among all the universal (i.e. Turing-complete) machines. Some works (e.g. [24]) show that it is impossible to find a canonical universal reference machine and any criterion to tell that a machine is better than another as a reference is arbitrary. However, other efforts (such as [1] or [32]) suggest that some machines (especially minimal machines) are a better choice than many other (more complex) machines. Additionally, the choice of minimal or simple machines has a rationale behind because it is generally easier to emulate simple machines than more complex machines and that the constants which make the difference will be generally smaller. As a conclusion of all this, we can see that we are forced to make some choices (as we will do in this paper), but we must realise that not every choice is equally good.

Another problem of using an arbitrary universal machine is that this machine can generate environments which are not discriminative. By discriminative we mean that there are different policies which can get very different rewards and, additionally, these good results are obtained by competent agents and not randomly. Note that if we generate environments at random without any constraint, we have that an overwhelming majority of environments will be completely useless to discriminate between capable and incapable agents, since the actions can be disconnected with the reward patterns, or there can be many situations where the actions have no influence on the rewards, with reward being good (or bad) independently of what the agent does.

In [6] a set of properties which are required for making environments discriminative are formally defined, namely that observations and rewards must be sensitive to agent's actions and that environments are balanced, i.e. that a random agent scores 0 in these environments (when rewards range from -1 to 1). This is crucial if we take time into account in the tests because if we leave a finite time to interact with each environments and rewards went between 0 and 1 a very proactive but little intelligent agent could score well. Given these constraints if we decide to generate environments without any constraint and then try to make a post-processing sieve to select which of them comply with all the constraints, we will have a computationally very expensive or even incomputable problem. So, the approach taken in this paper is to generate an environment class that ensures that these properties hold. But, we have to be very careful, because we would not like to restrict the reference machine to comply with these properties at the cost of losing their universality (i.e. their ability to emulate or include any computable function).

And finally, we would like the environment class to be user-friendly to the kind of systems we want to be evaluated (humans, non-human animals and machines), but without any bias in favour or against some of them.

According to all this, in this paper we present an operational way to define a universal environment class from which we can effectively generate valid environments, calculate their complexity and consequently derive their probability.

The rest of the paper is organised as follows. Section 2 discusses the general (and difficult) problem of generating a universal environment class which is valid for intelligence testing. Section 3 faces the problem by separating the notion of space from the notion of behaviour and objects. It also analyses the interface. Section 4 introduces the environment class and shows some examples. Section 5 proves that the class only includes environments which are observation and reward-sensitive and are balanced. Section 6 addresses the problem of automatically generating and coding the environments in order to construct the tests. And Section 7 closes the paper with a wrap-up discussion and the future work.

2 Turing-complete Environment Classes

Defining an environment class and constructing an environment generator from this class is easy. In fact, there are many environment and game generators, which construct different playgrounds following some basic meta-rules (e.g. [26]). A different issue is if we want to select an environment class which is universal (i.e. Turing-complete, meaning that any behaviour can take place, according to Church-Turing thesis) and it is unbiased (not particularly easy for some kind of agents and difficult for others). The problem is especially cumbersome because we are not only talking about a language which is able to express such environments, but a language which can be used to automatically generate such environments. Think, for instance, that we can generate random programs in any programming language, but the probability of any of these programs to make some sense is ridiculous.

Before getting into details, let us give a definition of environment. Following [20] and [6] an environment is a world where an agent can interact through actions, rewards and observations as seen in Figure 1.

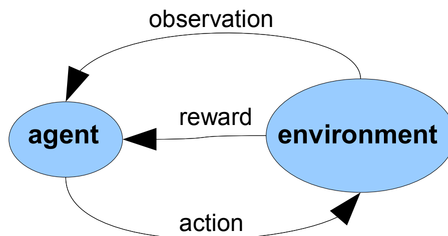

Fig. 1. Interaction with an Environment

Actions are limited by a finite set of symbols A, (e.g. $\{left, right, up, down\}$), rewards are taken from any subset R of rational numbers between -1 and 1, and

observations are also limited by a finite set O of possibilities (e.g., the contents of a grid of binary cells of $n \times m$, or a set of light-emitting diodes, LEDs). We will use a_i, r_i and o_i to (respectively) denote action, reward and observation at interaction (or, more loosely, state) i. The order of events is always: reward, observation and action. A sequence of two interactions is then a string such as $r_1 o_1 a_1 r_2 o_2 a_2$. Both the agent and the environment are defined as a probabilistic measure. For instance, given an agent, denoted by π, the term $\pi(a_k | r_1 o_1 a_1 r_2 o_2 a_2 \ldots r_k o_k)$ denotes the probability of π executing action a_k after executing the sequence of events $r_1 o_1 a_1 r_2 o_2 a_2 \ldots r_k o_k$. In a similar way, an environment μ is also a probabilistic measure which assigns probabilities to each possible pair of observation and reward. For instance, $\mu(r_k o_k | r_1 o_1 a_1 r_2 o_2 a_2 \ldots r_{k-1} o_{k-1} a_{k-1})$ denotes the probability in environment μ of outputting $r_k o_k$ after the sequence of events $r_1 o_1 a_1 r_2 o_2 a_2 \ldots r_{k-1} o_{k-1} a_{k-1}$. Note that if all the probability (1) is given to one action/output and 0 to the rest, we then have a deterministic agent/environment. Unless stated, in the following we will refer to deterministic agents/environments.

According to [6], the environments must be sensitive to observations and rewards (informally, they have to react to actions in such a way that eventually observations and rewards may be different). This just implies that the agent can have an effect on the environment and on its rewards. Additionally, environments must be balanced, in such a way that the expected value of the accumulated reward of a random agent (an agent choosing among possible actions at random) is 0.

Let us see a very simple environment:

Example 1. Consider a test setting where a robot (the agent) can press one of two possible buttons ($A = \{B_1, B_2\}$), rewards are just a variable score ($R = [-1, 1]$) and the observation is two cells where a ball must be inside one of them ($O = \{C_1, C_2\}$). Given the sequence of events so far is $r_1 o_1 a_1 r_2 o_2 a_2 \cdots r_{k-1} o_{k-1} a_{k-1}$, we define the environment behaviour as follows:

- If ($a_{k-1} = B_1$ and $o_{k-1} = C_1$) or ($a_{k-1} = B_2$ and $o_{k-1} = C_2$) then we generate a raw reward of $+0.1$.
- Otherwise the raw reward is -0.1.

The accumulated reward ρ of an agent is the sum of all the raw rewards produced so far. Additionally, if the agent reaches a state with raw reward r, then the returned reward is r unless $\rho + r > 1$ (where $1 - \rho$ is returned) or $\rho + r < -1$ (where $-1 - \rho$ is returned). In this way, we ensure that the accumulated reward ρ is always between -1 and 1.

The observation o_k in both cases above is generated with the following simple rule: if k is even then $o_k = C_2$. Otherwise, $o_k = C_1$. The first reward (r_1) is 0. It is easy to prove that the previous environment is observation and reward sensitive and it is also balanced.

From the previous example, a robot always pressing button B_1 would have the following interaction: $0 C_1 B_1 0.1 C_2 B_1 - 0.1 C_1 B_1 0.1 \ldots$ and an expected accumulated reward 0. A random agent (pressing buttons B_1 and B_2 at random) would also have an accumulated reward 0.

From the previous simple example we can see the many decisions we have to make about an environment. How many actions? How many observations? Note that in the previous example, these were minimised: only two possible actions and only two possible observations. And once this is determined, infinitely many options appear on the environment's behaviour.

The type and expressiveness of environments is a typical issue in artificial intelligence. For instance, [31] is a survey of environments used in multi-agent systems. In this survey, we can see many different ways in which actions and observations can be expressed. However, no model of computation is thoroughly described in any of them, so these descriptions at a shallow level give us no hint about how to construct a universal environment class.

Dramatically opposite is to go as general as possible and use Turing machines themselves. For instance, Persistent Turing Machines (PTMs) [4] can be a choice. PTMs are an extension of Turing machines to incorporate interaction, which basically boils down to having an extra tape used as memory between computations. So each macro-state of this machine receives an input (observation) and produces an output (action)[2]. Both observations and actions are strings of arbitrary length, so the number of possible observations and actions is unlimited. However, apart from being one possible way of modelling interaction, this approach is too broad in many senses: making a filter such that every computation between macro-states is finite is a difficult (more precisely, undecidable) problem. Additionally, the interface (observations and actions) is awkward (not user-friendly) for many agents (since a sequence of 0s and 1s, or squares and crosses), is not practical for many intelligent systems (including many people).

These general models of computation based on Turing machines are interesting and useful in a formal way, and can be used to define taxonomies of environments, such as the simple taxonomy that appears in [18] or the more elegant taxonomy found in [19]. However, these taxonomies are still too incipient to present an environment class which is suitable for our purpose.

Apart from Turing machines (or variants) there are many other Turing-complete models of computation, such as λ-calculus, combinatory logic, register machines, abstract state machines, Markov algorithms, term-rewriting systems, etc. Many of them are more natural and easy to work with than Turing machines. In fact, in [7] and [8], for measuring sequential inductive ability, we used a type of register machine.

From all the previous models, state machines are the closest formalism to the idea of an input-output environment. For instance, the following finite-state machine in Figure 2 represents an environment in a very easy way.
In the previous automaton, the set of possible actions is $\{a, b\}$ and the set of observations is $\{S, T, U\}$. An easy way to assign rewards is to set a reward value for each state, for instance 0 for S, $+1/10$ for T, and $-2/10$ for U.

Finite-state machines are, however, not Turing-complete. In fact, it is well-known that they are equivalent in expressiveness to regular languages (Type-3,

[2] If we see a persistent Turing machine as an environment then inputs are actions from the external agent and outputs are observations to the agent.

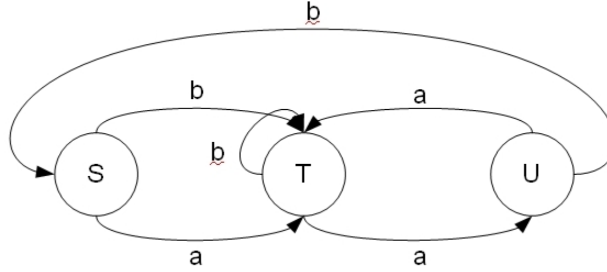

Fig. 2. A Simple Finite-State Automaton Modelling an Environment

in Chomsky's hierarchy) while Turing machines are in Type-0. In fact, if we restrict the environment class to a non-universal environment class (or more drastically to finite environment classes), we not only may have a strong bias but we can also have some other undesirable properties (see e.g. [14], where an environment class including a bounded set of finite state machines is shown to be useless to measure intelligence, since all the agents get the same score if all the environments are weighted equally, a phenomenon which is well-known in computer science and is known as 'No Free Lunch').

Linear-bounded and nondeterministic pushdown automata are, respectively, Type-1 and Type-2, while they lose the nice properties for representing environments. But, at type 0 (i.e. Turing-complete), we find Markov algorithms [23] (a kind of pattern-matching term-rewriting system without variables) which can easily be used to represent environments. Let us recover their definition:

Definition 1. *A Markov algorithm is a triple* $\langle \Sigma, \Gamma, \Pi \rangle$ *where* Σ *is an input alphabet,* Γ *is a work alphabet with* $\Sigma \subseteq \Gamma$ *and* Π *is an ordered sequence of rules (also called productions) of the form* $LHS \rightarrow RHS$ *(normal rules) or* $LHS \rightarrow\cdot\ RHS$ *(terminating rules) where* LHS *and* RHS *are (possibly empty) words in* Γ^*.

The algorithm is able to process an input string s *in* Γ^* *as follows:*

1. *Check the rules in order from top to bottom to see whether any of the strings to the left of the arrow can be found in* s. *If a rule with an empty string to the left is reached (this is usually at the bottom and is usually known as the default rule), it means that it necessarily has to be applied at the rightmost part of the input string.*
2. *If none is found, stop.*

3. *If one or more is found, replace the rightmost*[3] *matching text in the symbol string s with the text to the right of the arrow in the first corresponding rule, making a new string s.*
4. *If the applied rule was a terminating one, stop.*
5. *Go to step 1.*

An example of a very simple environment using a Markov algorithm is as follows:

Example 2. Consider two possible actions ($A = \{l, r\}$), rewards are just a variable score ($R \in \{-, +\}$) and the observation is a set ($O = \{w, b\}$). The behaviour of the environment is given by the following Markov algorithm, where $\Sigma = A \cup R \cup O \cup \{|, @\}$, with $|$ being a symbol to separate the interactions (except from the last interaction which is ended by @). We consider $\Gamma = \Sigma$.

1. $wl@ \rightarrow \cdot \ wl| + b@$
2. $bl@ \rightarrow \cdot \ bl| - w@$
3. $wr@ \rightarrow \cdot \ wr| - b@$
4. $br@ \rightarrow \cdot \ br| + w@$
5. $\rightarrow \cdot \ +b@$

The input string is the sequence of events so far, (i.e $r_1 o_1 a_1 | r_2 o_2 a_2 | \cdots | r_{k-1} o_{k-1} a_{k-1} @$) and the new reward and observation are given by the rightmost character in the resulting string which is in R and the rightmost character in the resulting string which is in O. The environment starts with the empty string and after each interaction the input string is parsed by the algorithm.

The behaviour of this environment can be explained as follows. It starts giving a positive reward (+) and observation b. This is the meaning of the last rule which is only executed when we start (the empty string). Then the environment outputs w and b alternately as can be seen in rules 1 to 4. And now, rewards go as follows. If the agent makes action l when observation is w then the reward is positive (+) as seen in rule 1. If the agent makes action l when observation is b then the reward is negative ($-$) as seen in rule 2. If the agent makes action r when observation is w then the reward is negative ($-$) as seen in rule 3. If the agent makes action r when observation is b then the reward is positive (+) as seen in rule 4.

Note that we can also define agents using this formalism:

Example 3. Consider the same sets A, R and O as in the previous example and the same use of $|$ as a symbol to separate the interactions except for the last interaction which ends with a @. In this case we can define a Markov algorithm for an agent, by using $\Gamma = \Sigma$ exactly equal as the previous example and the production rules:

[3] This is typically leftmost, but we change this because our strings representing environments interactions usually grow on the right, and the rightmost part of the strings is then more important. This is completely symmetric, so changing this is just a convention which does not change the power of these machines.

1. $w@ \rightarrow\cdot\ wl@$
2. $b@ \rightarrow\cdot\ br@$

where the input string is the sequence of events so far is $r_1o_1a_1|r_2o_2a_2|\ldots|r_{k-1}o_{k-1}@$ and the new action is given by the rightmost character in the resulting string which is in A.

The behaviour of this agent is very simple. If the observation is w then it makes action l. Otherwise it makes action r. Note that this agent is optimal with the previous environment (it always gets positive rewards (+)).

After the previous examples on environments and agents, it seems that just by generating random Markov algorithms we can get a pool of environments to use in our tests. This is not so easy, though.

The problem of using a universal (Turing-complete) language is that many production sets will not make up a valid environment. In fact, the very simple environment from Example 2 requires around 100 bits to be coded (more or less depending on the coding). Most of the other 2^{100} production sets of equal or lower complexity are not valid environments. They may produce an invalid sequence of rewards, observations and actions, or no sequence at all, or very short sequences (so the environment stops interacting very soon) or they do not terminate. Additionally, considering only the syntactically valid and terminating environments, there are other constraints imposed by the setting in [6]: they must be reward and observation sensitive and they must be balanced with respect to random agents. And, finally, there is also a certainty that in this way it is very unlikely to generate an environment which is social, i.e. that may contain other agents to interact with. As it is frequently argued (see e.g. [6]) animal (and consequently human) intelligence must have evolved in real environments full of objects and other animals.

Our approach then will be different. Instead of generating environments from uncontrolled universal machines, we will attempt to define a controlled set of actions and observations, and then we will include universal behaviours inside. We will approach this next.

3 On Actions, Observations and Space

Apart from the behaviour of an environment, which may vary from very simple to very complex, we must first clarify the *interface*. How many actions are we going to allow? How many different observations? We asked these questions above but we have not given an answer. The very definition of environment makes actions a finite set of symbols and observations also a finite set of symbols. It is clear that the minimum number of actions has to be two, but no upper limit seems to be decided a priori. The same happens with observations. Even choosing two for both, a sequence of interactions can be as rich as the expressiveness of a Turing machine, which only uses 0 and 1. In order to make this more user-friendly, we can, of course, use two buttons as actions and a single light bulb for observations.

If we take a look at some cognition tests for humans and non-human animals, the actions can be quite varied. For humans, it can be writing a word or a number following a sequence (note that the number of bits required to code a word or a number translates into a relatively big number of discrete actions). For animals, it is typically more reduced, with a short number of buttons to press or cups to raise [13]. Observations can have a much higher variation. For instance, a sequence of squares and symbols, or a sequence of numbers or words, or even images, are typical in IQ tests. These require many bits. Even in animal testing, we can have dynamic observations (rotations, cup movements, sounds, ...). All this takes a lot of bits and makes any discrete coding of the observation rather difficult. Typically, even for discrete environments, observations get large by a combinatorial combination. For instance, if we have five light bulbs which can glow with four different colours, we have 20 possible combinations if only one can glow at a time. This would require 5 bits of information. Or consider, for instance, the possible legal arrangements of pieces in a chess board.

Our choice is to make this number as small as possible and, in any case, dependent to the complexity of the environment. Simple environments should have simple interfaces. Complex environments might have simple or large interfaces.

Before going into detail about the interface, we have to think about environments that can contain agents. This is not only the case in real life (where agents are known as inanimate or animate objects, animals among the latter), but also a requirement for evolution and, hence, intelligence as we know it. The existence of several agents which can interact requires a *space*. The space is not necessarily a virtual or physical space, but also a set of common rules (or laws) that govern what the agents can perceive and what the agents can do. From this set of common rules, specific rules can be added to each agent. In the real world, this set of common rules is physics. All this has been extensively analysed in multi-agent systems (see, e.g., [18] for a discussion).

The good thing about thinking of spaces is that a space entails the possible perceptions and actions. If we define a common space, we have many choices about observations and actions already made.

A first (and common) idea for a space is a 2D grid. This is the typical choice in mazes and 2D videogames. Additionally, it eases the test interface when using 2D screens. From a 2D grid, the observation is a picture of the grid with all the objects and agents inside. In a simple grid where we have agents and objects inside the cells, the typical actions are the movements left, right, up and down. Alternatively, of course, we could use a 3D space, since our world is 3D. In fact, there are some results using intelligence testing (for animals or humans) with a 3D interface [25][30].

The problem of a 2D or 3D grid is that it is clearly biased in favour of humans and many other animals which have hardwired abilities for orientation in this kind of space. Other kinds of animals or handicapped people (e.g., blind people) might have some difficulties in this type of spaces. Additionally, artificial intelligence agents would benefit highly by hardwired functionalities about

Euclidean distance and 2D movement, without any real improvement in their general intelligence.

Instead we propose a more general kind of space. A 2D grid is a graph with a very special topology, where there are concepts which hold such as direction, adjacency, etc. A generalisation is a graph where the cells are freely connected to some other cells with no particular pre-defined pattern. For instance, if we consider S, T and U as cells in Figure 2, we have a space with three cells and some possible movements that lead from cell to cell.

Figure 3 also shows two more regular spaces. Note that the actual proximity of two cells (in terms of arrows between them) do not generally correspond to their proximity in a 2D representation (even in these regular spaces)[4].

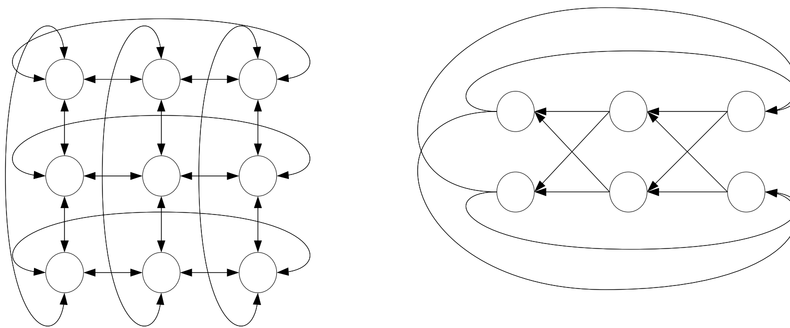

Fig. 3. Some example spaces. Left: a toroidal grid. Right: A two-level ring.

So, this suggests a (generally) dimensionless space. Connections between cells would determine part or all of the possible actions. For instance if a cell 3 has three arrows to cells 4, 5, and 6, then if an agent is at 3 then the possible actions could include going to 4, 5 or 6. Other actions (such as catching an object in the cell or posting a message) would also possible if so defined. Additionally, it is not necessary to include all the possible arrows as actions. For instance, if an agent is at 3 then the possible actions could be going to 4 or 5 but not to 6. Consequently, a space sets a common playground for the agents, but there are still many options for actions.

The observations and actions could be shown graphically as seen in Figure 4, which illustrates two possible interfaces. Note that cells are not distributed by their actual proximity, but just in a horizontal line. The cell with the thickest frame is the current cell, and the cells which are pointed by the arrows are the cells where the agent can move to. The agents and objects are represented by different symbols. Depending on the agent (e.g. a human adult or a chimpanzee),

[4] In irregular spaces, a possibility is to apply multidimensional scaling to place the spaces in 2D. This would give an advantage to agents which have a predefined sense of Euclidean distances in 2D.

this could be turned into a kind of tactile interface (where cells can be pressed) or a more robust one (with buttons). For artificial agents, it really does not matter.

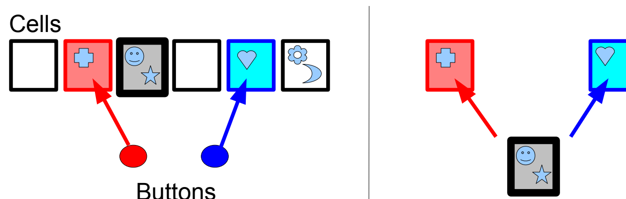

Fig. 4. Possible interfaces for an environment based on an adimensional space. Left: all the cells are shown. Right: only current and reachable cells are shown

Note that not all the connections between cells are explicit (we do not see the complete graph, only the part which is connected to the current cell). This implies (as do many other parts of the environment) that the agent requires memory to learn the connections (and also inductive inference abilities if the connections have a pattern). We can further restrict the observation to only the current cell and the cells where the actions lead (right of Figure 4). This would be problematic for some agents (e.g. small children and some animals) for whom what you cannot see does not exist. In the end, there are many different choices for the interface and we have to be very careful if we do not want to introduce a bias. A possible suggestion here is to follow the expertise and tradition of other fields, such as comparative psychometrics (see, e.g., [13]).

4 Definition of the Environment Class

After the previous discussion, we are ready to make some important choices and give the definition of the environment class. First we must define the space and objects, and from here observations, actions and rewards. Before that, we have to define some constants that affect each environment. Namely, with $n_a = |A| \geq 2$ we denote the number of actions, with $n_c \geq 2$ the number of cells, and with n_ω the number of objects/agents (not including the agent which is to be evaluated and two special objects known as Good and Evil).

4.1 Space

The space is defined as a directed labelled graph of n_c nodes (or vertices), where each node represents a cell. Nodes are numbered, starting from 1, so cells are refered to as $C_1, C_2, \ldots, C_{n_c}$. From each cell we have n_a outgoing arrows (or arcs), each of them denoted as $C_i \to_\alpha C_j$, meaning that action $\alpha \in A$ goes from C_i to C_j. All the outgoing arrows from C_i are denoted by (C_i). At least two

outgoing arrows cannot go to the same cell. Formally, $\forall C_i : \exists r_1, r_2 \in (C_i)$ such that $r_1 = C_i \rightarrow_{\alpha_m} C_j$ and $r_2 = C_i \rightarrow_{\alpha_n} C_k$ with $C_j \neq C_k$ and $\alpha_m \neq \alpha_n$.

At least one of the outgoing arrows from a cell must lead to itself (typically denoted by α_1 and is the first action). Formally, $\forall C_i : \exists r \in (C_i)$ such that $r = C_i \rightarrow_{\alpha_1} C_i$.

A path from C_i to C_m is a sequence of arrows $C_i \rightarrow C_j, C_j \rightarrow C_k, \ldots, C_l \rightarrow C_m$. The graph must be strongly connected, i.e., all cells must be connected (i.e. there must be a walk over the graph that goes through all its nodes), or, in other words, for every two cells C_i, C_j there exists a path from C_i to C_j and viceversa. Note that this implies that every cell has at least one incoming arrow.

Given the previous definitions, the topology of the space can be quite varied. It can include a typical grid, but much more complex topologies, too. In general, the number of actions n_a is a factor which influences the topology much more than the number of cells.

Note that because at least one arrow must be reflexive and at least another has to go to a different cell, then if $n_a = 2$, then the space is necessarily a ring of cells.

4.2 Objects

Cells can contain objects from a set of pre-defined objects Ω, with $n_\omega = |\Omega|$. Objects, denoted by ω_i, can be animate or inanimate, but this can only be perceived by the rules each object has. An object is inanimate (for a period or indefinitely) when it performs action α_1 repeatedly. Objects can perform actions following the space rules, but apart from these rules, they can have any behaviour, either deterministic or not. Objects can be reactive and can be defined to act with different actions according to their observations. Objects perform one and only one action at each interaction of the environment (except from the special objects Good and Evil, which can perform several actions in a row).

Apart from the evaluated agent π, as we have mentioned, there are two special objects, called Good and Evil[5], represented by $\oplus$ and $\ominus$ respectively, when they are seen by the evaluated agent π. However, they are indistinguishable for the rest of objects (including Good and Evil themselves), so for them in their observations they are represented by the same symbol $\odot$.

Good and Evil must have the same behaviour. By the *same* behavior we do not mean that they perform the same movements, but they have the same *logic* or *program* behind them. Note that Good and Evil see each other in the same way (and so do the rest of objects except π). For instance, if Good has the program "do action α_2 unless $\odot$ is placed in the cell where α_2 leads. In this case, do action α_1" then Evil should have the program "do action α_2 unless $\odot$ is placed in the cell where α_2 leads. In this case, do action α_1". For the first, $\odot$ refers to Evil and for the second $\odot$ refers to Good.

[5] The need of both Good and Evil will be seen later on with the property of balancedness.

Objects can share a same cell, except Good and Evil, which cannot be at the same cell[6]. If their behaviour leads them to the same cell, then one (chosen randomly with equal probability) moves to the intended cell and the other remains at its original cell. Because of this, the environment becomes stochastic (non-deterministic).

Objects $\oplus$ and $\ominus$ can execute several actions in a single interaction, i.e. they can perform any (non-empty) sequence of actions. One reason for this is to avoid Good being followed by the agent as an easy and optimal way to get positive rewards in most environments.

Objects are placed randomly at the cells with the initialisation of the environment. This is another source of stochastic behaviour.

Although Good and Evil have the same behaviour, the initial cell which is (randomly) assigned to each of them might determine a situation where their behaviour is finally very asymmetric wrt. agent π. For instance, consider the following example:

Example 4. Imagine the following space shown in Figure 5 and consider the behaviour of $\oplus$ and $\ominus$ in such a way that they perform action r if and only if the agent π shares a cell with any of them (note that this can be perceived in the observations) or $\oplus$ or $\ominus$ are on the left. Otherwise, they perform action l.

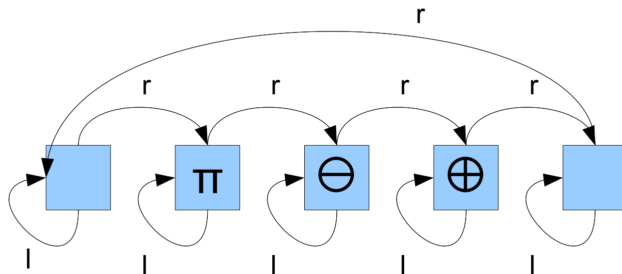

Fig. 5. A ring space where the initial state can be critical.

From the state depicted in Figure 5, it is easy to see that the relative situation of the three objects can only be as shown (π, $\ominus$, $\oplus$) or when the agent shares a cell with $\ominus$, followed by $\oplus$ on the right. Consequently, in this environment, it is impossible for π to share a cell with $\oplus$, while it is possible with $\ominus$, even though $\oplus$ and $\ominus$ have the same behaviour. The initial state is critical.

Following the previous example, we can define a "cycle clause" which works as follows. Given an environment with n_a actions, and n_c cells, we calculate a

[6] This condition could be relaxed but it is introduced in order to prevent from having both agents doing the same actions forever.

random number n between 1 and ${n_c}^{n_a}$ (uniformly), and then after n interactions, the positions of $\oplus$ and $\ominus$ are swapped. Then, we compute another random number again (in the same way) and then positions will be swapped again. And so on. The rationale for being random is to prevent the cycle from matching any cycle or pattern which is present in the behaviours of the objects, such as "go close to π until interaction n, when you must change your behaviour and run away indefinitely". The goal of this clause is to avoid the relevance of the initial state in the environment, because it can be critical.

Finally, the first interactions with an environment can have what we call "start-up garbage" [12]. Consider, e.g., a behaviour for $\oplus$ and $\ominus$ which is "start doing $a_1a_2a_0a_1a_1a_1a_0a_2a_2a_0a_1a_1a_0a_0$ and then do a_0a_1 for ever". The first part of their behaviour is completely random and is completely non-discriminative. Only when the pattern is reached (the second part of the behaviour), it makes sense to start evaluating the behaviour of an agent. Consequently, we suggest letting a random agent play for n interactions in order to surpass most of the start-up garbage (if it exists) and then start the evaluation. The value for n can be made equal to the previous value used in the "cycle clause".

4.3 Observations and Actions

The observation is a sequence of cell contents. The cells are ordered by their number. Each element in the sequence shows the presence or absence of each object, including the evaluated agent. Additionally, each cell which is reachable by an action includes the information of that action leading to the cell. In particular, the content of each cell is a sequence of objects, where π must appear before $\oplus$ and $\ominus$, and the rest of the objects following their index. Then the possible actions follow, also ordered by their index, and denoted by A_i instead of α_i. The reason for using a different alphabet in the observations will be seen later on in the paper. Each cell content sequence is separated by the symbol ':'.

For instance, if we have n_a=2, n_c=4 and n_ω=2 then the following sequence $\pi\omega_2A_1 : \ominus : \oplus\omega_1A_2 ::$ is a possible observation seen by the evaluated agent π. The meaning of this sequence is that at cell 1 we have the evaluated agent and object ω_2, at cell 2 we have Evil, at cell 3 we have Good and object ω_1 and cell 4 is empty. Additionally, we see that we can stay at cell 1 with action α_1 and we can go to cell 3 with action α_2. The same observation would be seen as $\pi\omega_2A_1 : \odot : \odot\omega_1A_2 ::$ by the rest of the objects (including Good and Evil[7]).

4.4 Rewards

Raw rewards are defined as a function of the position of the evaluated agent π and the positions of $\oplus$ and $\ominus$. If we only give rewards when they are at the same cell, we may have environments where $\oplus$ and $\ominus$ move faster than π and no reward at all can be obtained. Consequently, a first idea is to use a

[7] There is still the question of whether Good and Evil can guess where they are, even though the use of this symbol.

quasimetric, i.e. to use the minimum number of actions to reach $\oplus$ and $\ominus$ to calculate two quasidistances. From these measures (positive for $\oplus$ and negative for $\ominus$), we could derive a value for the raw reward. A problem with this approach is that some environments (such as the space on the right of Figure 3) can have situations where both quasidistances are equal forever and the environment always produces reward 0. In the space on the right of Figure 3, imagine that $\oplus$ and $\ominus$ are located at the same separation from π (one always located at the upper cell and the other at the lower cell). If their movements always maintain a certain separation w.r.t. agent π the distances will always be the same and rewards will always be 0.

Instead of that, we will work with the notion of trace and the notion of "cell reward", which we denote by $r(C_i)$. Initially, $r(C_i) = 0$ for all i. Cell rewards are updated by the movements of $\oplus$ and $\ominus$. At each interaction, we set $r_i^{\oplus}$ to the cell reward where $\oplus$ is and $-r_i^{\ominus}$ to the cell reward where $\ominus$ is. Each interaction, all the other cell rewards are divided by 2. So, an intuitive way of seeing this is that $\oplus$ leaves a positive trace and $\ominus$ leaves a negative trace. The agent π *eats* the rewards it finds in the cells it occupies. We mean it *eats*, since just after getting the reward, the cell reward is set to 0. If either $\oplus$ or $\ominus$ arrive at a cell where π was, the reward is stays at 0 while both objects share the cell. In other words $\oplus$ and $\ominus$ give reward 0 if they share a cell with π.

The values of $r_i^{\oplus}$ and $-r_i^{\ominus}$ that $\oplus$ and $\ominus$ leave on the rest of the occasions are also part of the behaviour of $\oplus$ and $\ominus$ (which is the same, but this does not mean that $r_i^{\oplus} = r_i^{\ominus}$ for every i). Only one constraint is imposed on how these values can be generated, $\forall i\ : 0 < r_i^{\oplus} \leq 1/2$ and $0 < r_i^{\ominus} \leq 1/2$. A typical choice in the following examples is just to make $r_i^{\oplus} = r_i^{\ominus} = 1/2$ for all i, although much more complex reward behaviours are needed to code some complex environments. Finally, note that rewards and traces are not part of the observations, so they cannot be (directly) observed by any object (including π).

When π moves to a cell, it gets the cell reward which is at that cell, i.e. the accumulated reward $\rho = \rho + r(C_i)$.

In order to maintain accumulated rewards between -1 and 1, one possibility is to saturate them inside this interval. By $Trunc_{-1}^{+1}(\rho)$ we denote that if $\rho > 1$ then ρ is set to 1, and if $\rho < -1$ then ρ is set to -1.

However, if we accumulate and saturate rewards, a problem appears if the agent is satisfied by a good accumulated reward and stopped by its complacency. In order to prevent π from resting on its laurels, there are some possibilities. One possibility is that, for each interaction, the accumulated reward ρ is divided by 2. This is called the degradation of the accumulated reward.

Another option is to calculate the average of the rewards, i.e. at the end of the evaluation, we divide the accumulated reward by the final number of interactions (denoted by n_i). The justification for this option is further investigated in [5].

With this sharing of rewards, the complete order of rewards, observations and actions is as follows:

1. $\rho = 0$
2. $\forall i\ :\ r(C_i) \leftarrow 0$

3. Place objects randomly such that $\oplus$ and $\ominus$ are in different cells.
4. An initial reward ρ is given to π.
5. $i \leftarrow 0$
6. $i \leftarrow i + 1$
7. For every C_j check whether it contains $\oplus$ and not π. If so, $r(C_j) \leftarrow +r_i^{\oplus}$.
8. For every C_j check whether it contains $\ominus$ and not π. If so, $r(C_j) \leftarrow -r_i^{\ominus}$.
9. Observations are produced for all the objects.
10. Object π acts and moves to a given cell (denoted by C_π).
11. Rest of objects act.
12. $\rho \leftarrow \rho + r(C_\pi)$.
13. $\forall j : r(C_j) \leftarrow r(C_j)/2$.
14. Until the stopping criterion is not met, go to 6.
15. $\rho \leftarrow \rho/i$

We will use the term Λ for this environment class.

4.5 Examples

Given the configuration of the environment class given above, let us see some examples inside Λ. Let us start with the simplest environment inside the class.

Example 5. Consider number of actions $n_a = 2 = |\{\alpha_1, \alpha_2\}|$, number of cells $n_c = 2$ and number of objects $n_\omega = 0$. The space is necessarily made of two cells and two arrows each, α_1 just going to the same cell and α_2 going to the other cell. The simplest behaviour for $\oplus$ and $\ominus$ is to execute α_1 forever or to execute α_2 forever, and $r_i^{\oplus} = r_i^{\ominus} = 1/2$ forever. Initially, evil and good are randomly located in the two cells (since they cannot be in the same cell). Since it is symmetric, consider that $\oplus$ is at cell 1 and $\ominus$ is at cell 2. In case the behaviour is α_1 (remember it is the same for both $\oplus$ and $\ominus$), then both Good and Evil will stay in their cells forever. The reward will then be 1/2 if the agent starts at the position where $\ominus$ is and moves to the place where $\oplus$ is, then getting 0 rewards because it stays where $\oplus$ is. If the agent starts at the position where $\oplus$ is then the move to the cell where $\ominus$ is will get $-1/2$ and soon it will try to go back where $\oplus$ is, getting 1/2. Although the average will converge to 0, it is convenient to stay where $\oplus$ is. In case the behaviour of both $\oplus$ and $\ominus$ is α_2, then both Good and Evil will swap cells forever observation after observation. Consequently, the best strategy for the agent will be to move to the cell where $\oplus$ is, getting 1/2 on average.

Let us see another more complex environment in this class:

Example 6. Consider $n_a = 3 = |\{s, l, r\}|$, $n_c = 5$ and $n_\omega = 1$ and the topology shown in the following Figure 6.

The behaviour of $\oplus$ and $\ominus$ is moving left to right and right to left in the space forever (i.e. l, l, l, l, r, r, r, r) forever, unless they have to go to a cell where ω_1 is or they are in a cell where π is. In these two cases they do not make the move. If ω_1 goes to the same cell where $\oplus$ is, then it moves one position in the opposite

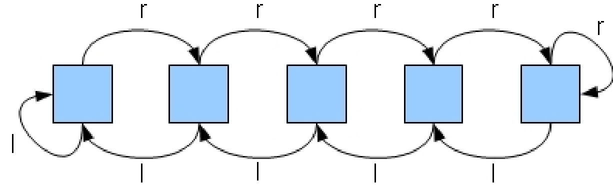

Fig. 6. The space of an environment with reactive objects and a more complex strategy

direction from where ω_1 came from or to the only position possible (apart from the same cell) in case of being in one of the edges of the space. The same for $\ominus$. The behaviour of ω_1 is to stay (i.e. s forever) unless π is in the same cell. In that case, ω_1 moves one position in the opposite direction from where π came from or to the only position possible in case of being in one of the edges of the space.

For instance, if we start as shown in Figure 7, we see that $\ominus$ is on the same side as π and rewards will be negative for π. We can wonder how to set ω_1 in a way such that it separates π from $\ominus$ while leaving $\oplus$ on the same side.

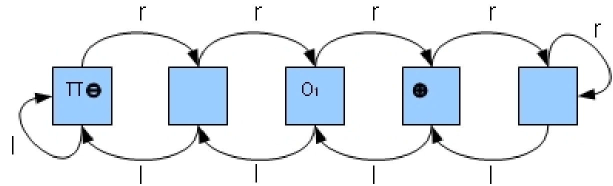

Fig. 7. A possible starting state.

And this is possible in a few moves as shown in Figure 8. From this moment on, if π just executes s all the time, the rewards will always be positive.

More complex environments can be generated. In fact, with some patience, any kind of game or task we know can be simulated using this environment class. The following example shows how a noughts-and-crosses (tic-tac-toe) game is inside Λ.

Example 7. Consider $n_a = 13 = |\{stay, goto_{1,1}, goto_{1,2}, goto_{1,3}, goto_{2,1}, goto_{2,2}, goto_{2,3}, goto_{3,1}, goto_{3,2}, goto_{31,3}, goto_{E1}, goto_{E2}, goto_{E3}\}|$, $n_c = 12$ and $n_\omega = 1$, where the

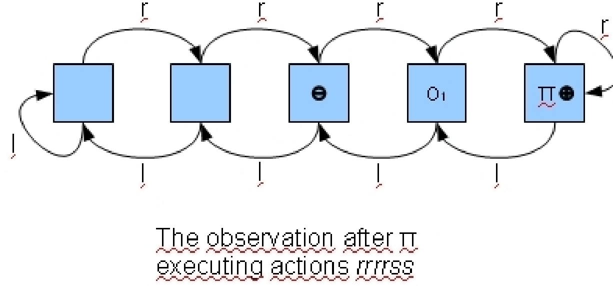

Fig. 8. A good situation for π if a good strategy is used.

first 9 cells are arranged in a 3×3 grid and there are other three extra cells outside this arrangement. Actions allow the agent to stay in a cell (*stay*), to move to any of the nine cells in the grids (action $goto_{X,Y}$ denotes cell $(X-1)*3+Y$) or to move to any of the three extra cells (by actions $goto_{E1}$, $goto_{E2}$ and $goto_{E3}$). Consequently, all the cells are directly connected to all other cells.

Object n_1 is the opponent and implements a reasonably good strategy for tic-tac-toe moving in the 3×3 grid (depending on its policy, we get different environments or levels). Objects $\oplus$ and $\ominus$ stay at cells number 10 and 11 (those to which $goto_{E1}$ and $goto_{E2}$ go respectively) initially (or viceversa). Initially, objects ω_1 and π are placed at the cell number 12 (the one which is reached with the action $goto_{E3}$).

The game goes as follows. At interaction 1, object ω_1 moves to one of the cells in the grid and π must stay at cell 12. At the next interaction, ω_1 moves to cell 12 and π can move to one of the cells in the grid not previously visited by ω_1. At the next interaction, π must execute one of the actions $goto_{E1}, goto_{E2}, goto_{E3}$ to leave the grid, and object ω_1 moves to one of the cells in the grid not previously visited by ω_1 or π. And so on. When the three-in-a-row is obtained, then both ω_1 and π must execute any of the $goto_{E1}$, $goto_{E2}$, $goto_{E3}$ actions and the game re-starts but now it is π who starts moving.

In order to distinguish between legal and illegal movements for π, and between winning or losing, rewards go as follow. If π makes an illegal move then $\oplus$ and $\ominus$ produce $r_i^{\oplus} = 0.01$ and $r_i^{\ominus} = 0.01$. If π makes a legal move then $\oplus$ and $\ominus$ produce $r_i^{\oplus} = 0.1$ and $r_i^{\ominus} = 0.1$. If π makes a three-in-a-row, than $\oplus$ and $\ominus$ produce $r_i^{\oplus} = 0.5$ and $r_i^{\ominus} = 0.5$. In order to get the rewards, π must go to either $E1$ or $E2$ when it is not its turn. Since π can distinguish between $\oplus$ and $\ominus$, in order to get the good rewards, it should do the following: stay in cell 12, make a move inside the grid, go to the cell where $\oplus$ is to get the reward, make a move inside the grid, go to the cell where $\oplus$ is to get the reward, etc.

The previous example shows how a two-player game can be turned into an environment in class Λ, using another object ω_1 as opponent and where $\oplus$ and $\ominus$ become judges or referees which stay still but assign rewards. It is like the jury in a competition. Since $\oplus$ can only give positive values and $\ominus$ only negative

values, then the agent should go to the cell where $\oplus$ is. Note that it is very easy to make this choice, but at least a random agent would get an expected reward equal to 0.

The issue now is first to prove that this environment class follows the constraints we require for a proper adaptive test (the anytime test presented in [6]) and then we will have to address the problem of automatically generating the environments.

5 Properties

In this section, we are going to prove that the previous environment class is sensitive to rewards and observations (the agent can perform actions in such a way that can affect the rewards and the observations), and it is also balanced (a random agent would have an expected accumulated reward equal to 0). For the formal definition of these properties, see [6].

Proposition 1. *The previous environment class Λ is observation-sensitive.*

Proof. The proof is trivial. The observation includes the position of π and π can move to different cells, because there are at least two cells, and the number of actions is equal or greater than 2, and at least one outgoing arrow must lead to different cells. Consequently, agent π can move from cell to cell in any occasion and consequently the observations change. □

The previous proposition has little relevance, because although the agent can move, this does not mean that rewards must be affected. The following proposition is more important then.

Proposition 2. *The previous environment class Λ is reward-sensitive.*

Proof. Since the graph is strongly connected, all cells are reachable by π. We have to proceed by cases, enumerating all four cases.

1. The cell C_j where π is has a non-zero reward, i.e. $r(C_j) \neq 0$. Then the agent can stay (there is always that action in any environment) setting the cell to 0. In this case, we go to cases 2, 3 or 4.
2. The cell C_j where π is has zero reward, i.e. $r(C_j) = 0$, and both $\oplus$ and $\ominus$ are in other different cells, let's say C_k and C_l. Then the agent has the following options. Stay forever, getting 0 reward forever (independently of whether $\oplus$ or $\ominus$ come to the cell, since they cannot give rewards if they are sharing a cell with π), or moving out. If it moves out there must be a path (the graph is completely connected) to both cells C_k and C_l. If it goes to C_k, many different things can happen in the meantime. If π gets any reward in this period, we then know that leaving C_j gets different reward than staying there. If it does not, then it arrives to C_k without any change in reward. But neither $\oplus$ nor $\ominus$ can set this cell C_k to 0 since for all i, $0 < r_i^{\oplus} \leq 1/2$ and $0 < r_i^{\ominus} \leq 1/2$. Consequently, π gets a positive or negative reward which is not 0. So, in this case, agent π can choose between a 0 reward and a non-zero reward.

3. The cell C_j where π is has zero reward, i.e. $r(C_j) = 0$, and $\oplus$ shares the cell with it and $\ominus$ is in a different cell C_k. Then the agent has the following options: (i) stay forever, getting 0 reward forever, or (ii) moving out. If it moves out there must be a path (the graph is completely connected) to cell C_k. Many different things can happen in the meantime. But neither $\oplus$ nor $\ominus$ can set this cell C_k to 0. Consequently, π gets a positive or negative reward which is not 0. So, in this case, agent π can choose between a 0 reward and a non-zero reward.
4. In the final case, the cell C_j where π is has zero reward, i.e. $r(C_j) = 0$, and $\ominus$ shares the cell with it and $\oplus$ is in a different cell C_k. This is symmetric to the previous case, case 3.

Since the four previous cases are exhaustive and all of them allow us to get different rewards depending on the action, then the environments are reward-sensitive. □

And finally, let us evaluate the balancedness condition.

Proposition 3. *The previous environment class Λ is balanced.*

Proof. By the definition of Λ, the raw reward is always in the interval $[-1, 1]$ and the accumulated reward ρ is also always in $[-1, 1]$ because we divide by the number of interactions. We also know that the special objects $\oplus$ and $\ominus$ have the same behaviour. The rest of objects (except π) cannot perceive the distinction between $\oplus$ and $\ominus$. Consequently, the chances of $\oplus$ and $\ominus$ being more favourable to π or behaving differently can only depend on two things: the agent π itself and the initial state. Since the initial state is chosen at random, for a random agent who chooses among its possible actions at random, the expected value of rewards for each cell is 0, since the probability of any action and state is equally probable as its symmetric state (everything equal except the positions and traces of $\oplus$ and $\ominus$, that would be swapped). Consequently, the expected accumulated reward is 0. □

Note that the previous proposition holds because of the random choice of the initial state. For instance, for Example 4 at the state shown in Figure 5, that initial state makes the expected reward for a random agent from then on not 0 (it is negative). Let us see whether we can obtain a stronger result which says that the environment is always balanced (and not only before the random choice of the initial object locations).

Proposition 4. *The previous environment class Λ is strongly balanced, i.e. at any interaction the expected accumulated reward of a random agent (from that interaction on) is 0.*

Proof. The proof is based on the previous one. We know that the expected accumulated reward for a random agent only depends on the initial state. Depending on this initial state, the expected value can be different from 0 without considering swapping. But, by the cycle clause, there is always an interaction where

$\oplus$ and $\ominus$ are swapped. Since the cycle length is chosen randomly and it is independent of the behaviour of $\oplus$ and $\ominus$, and the agent behaves randomly (and hence independently of their behaviour and the swaps), then we have that at any interaction j we can consider a sequence of raw rewards $r_j, r_{j+1}, \ldots$, ignoring swappings. If all of them are 0, then the proposition is proven. If the application of the accumulated reward function to them gives a positive value p, it is because there are more positive than negative in the limit (since the mean of the accumulated value). Consequently, every future swap will give a value $-p$ (ignoring all the further future swappings). Reiteratively, we have an infinite sequence of p, $-p$, p, $-p$, ..., whose expected value is 0. We reason similarly when the expected value is initially negative. □

The previous propositions show that the environments follow the properties required for their use in an anytime test [6]. This is important, since we want environments to be discriminative (and sensitivity is an important condition for that) and we want them to be unbiased w.r.t. rewards (i.e., we do not want random agents to behave well).

Another property which is not so relevant for testing but is interesting from a theoretical point of view (and especially when trying to compare some abilities) is whether the previous environment class allows 'perceptual aliasing'. The answer is a clear yes. Given the complex behaviours of objects, even a complete observation of the space is not sufficient (even knowing the behaviour of objects) since we do not know their internal state or memory.

6 Environment Complexity, Coding and Generation.

Finally, in order to construct a test, we need to generate the environments automatically and calculate their complexity. In order to generate an environment, we need to generate the space (constants and topology) and the behaviour for all the objects (except π). The first idea can be the use of a generative grammar, as usual. At least, this seems easy for the space. There are generative grammars that construct graphs, for instance. However, if we are going to use a generative grammar to code the space (and hence calculate its complexity), then we could choose a generative grammar which is not universal but only generates valid spaces. This is a mistake. With a non-universal generative grammar, it might happen that the complexity of a space with 100 identical cells could be 100 times the complexity of a space with one cell. But this is completely against the notion of Kolmogorov complexity and the ordering of environments that we are looking for. Note that the probability of an environment e must be based on $K(e)$ and not on its length. A regular 100x100 grid is generally simpler than a more intricate space with a much smaller number of cells.

Instead of that, we will use universal generative grammars, a Markov algorithm in particular. Before giving a general definition on how to code and generate environments, we are going to illustrate how Markov algorithms can code environments in class Λ with an example. For instance, the coding of example 5 is as follows:

- Code the Markov algorithm which generates the space, which is:
  1. $S \rightarrow +r$
  2. $\rightarrow SS\Omega$
  3. $\Omega \rightarrow \cdot$

  Note that this generates $+r+r$, which is interpreted as the first cell has an arrow with action l which goes to cell + 0 (i.e. itself and hence it does not appear), then the action r goes to cell +1 (+r), and the second cell is exactly the same. Since there are only two cells, cell 2 + 1 = cell 1.
- Code the objects. Only Evil/Good, with the following Markov algorithm:
  1. $\rightarrow \cdot\ l$

  Note that this behaviour of Evil/Good is not reactive. It just outputs l forever.
- Rewards do not need to be coded, since they are produced following the same rules for all the environments in class Λ.

Having seen the previous simple example, now we can give the definitions in general next.

6.1 Coding and Generating Spaces

First we code the number of actions n_a using any standard coding for natural numbers (e.g. the function $\log^*$ in [27][29]). The space graph is defined by a Markov algorithm with no constraints on their definition, but the following post constraint. The generated space has to be defined by a string as follows (we use regular language notation) $[\{+|-\}a_1^+][\{+|-\}a_2^+]\ldots[\{+|-\}a_{n_a}^+]$ for each cell. That means that we enumerate all the cells, and the information in each cell is composed of all the outgoing arrows (more precisely to which cells they go by the number of times the action appears). We use a toroidal indexing where, e.g. $1-2 = n_c - 1$, so we can use positive or negative references (this is the meaning of the $+|-$). When we refer to the same cell the action is omitted.

The space of Figure 5 is coded by $+r+r+r+r+r$ and a short Markov algorithm is given by:

1. $S \rightarrow +r$
2. $\rightarrow SSSSS\Omega$
3. $\Omega \rightarrow \cdot$

The space of Figure 6 is coded by $+r-l+r-l+r-l+r-l$ and a short Markov algorithm is given by:

1. $S \rightarrow +r-l$
2. $T \rightarrow SS$
3. $\rightarrow TT\Omega$
4. $\Omega \rightarrow \cdot$

With the previous description and definitions it is not difficult to see how to *code* (not univoquely) any valid environment. However, if we want to *generate* environments using Markov algorithms, the thing is much more difficult, since a random generation of Markov rules can generate thing such as this:

1. $S \rightarrow^{\cdot} +-++$
2. $\rightarrow SrST$

which does not represent any environment. Although optimisations might exist, since we do not want to lose generality and mangle the measurement of complexity, we propose here to let the algorithm run for a limited number of iterations and then pass a postprocessing (eliminate repeated $+$, invalid symbols, etc.) and then check whether the resulting string is a valid environment (syntactically and semantically, i.e. completely connected).

For the complexity calculation, any approximation of the length of the Markov algorithm (e.g. the number of symbols of the whole algorithm) would be valid as an approximation to its complexity.

6.2 Coding and Generating Objects

First we code the number of objects n_ω using any standard coding for natural numbers as mentioned above. Then we have to code their initial cell, by also coding $n_\omega + 3$ natural numbers bounded by n_c (not necessarily different, consequently $\log n_c$ each). In case $\oplus$ and $\ominus$ are in the same cell, a new cell from all the others is also generated randomly and assigned to $\ominus$.

The behaviour (which must be universal) is generated by another Markov algorithm as follows. The input string of the algorithm is the observation, as discussed in previous sections. With only the current observation the objects would not have access to memory and would be very limited. In order to overcome this limitation, two special symbols Δ and ∇ mean to put the current string into memory or to recover the last inserted string from memory respectively. Note that this history is the memory[8] and is necessary for complex behaviours. Once the Markov algorithm is executed for a fixed period of time (time-bounded), all symbols which are not in the set of actions A are removed from the output string. Then, for all the objects (except $\oplus$ and $\ominus$), the rightmost action in the string is the action to be made. For $\oplus$ and $\ominus$ the whole string is used.

Let us see an example of the behaviour for $\oplus$ and $\ominus$ (remember that they are equal) for the space shown in Figure 5 and considering no other objects:

1. $\pi[\odot]L \rightarrow^{\cdot} rrr$
2. $\rightarrow^{\cdot} l$

The notation [ ] represents an optional part. For instance, $\pi[\odot]L \rightarrow^{\cdot} rrr$ means in fact a pair of rules $\pi L \rightarrow^{\cdot} rrr$ and $\pi\odot L \rightarrow^{\cdot} rrr$. Consequently, the previous algorithm means to always perform action l unless π is found immediately on the left. In this case, we perform action r three times. For instance, if the observation in Figure 5 for $\ominus$ is represented by $: \pi L : \odot : R\odot ::$, then the application of the previous algorithm on that observation gives as result $: rrr\odot : \odot : R\odot ::$ and after the postprocessing (all the symbols which are not in the set of actions A are

[8] An alternative would be to use an internal string as local memory for each object. In any case, note that this is related to Persistent Turing Machines [4]

removed) we would have rrr. Note that only $\oplus$ and $\ominus$ can perform a sequence of more than one action.

And now let us see a more complex behaviour for object ω_1 with the space of Figure 6.

1. $\pi[\odot]\omega_1 \rightarrow\ M$
2. $M \rightarrow\ \nabla$
3. $\pi[\odot]L : [\odot]\omega_1 \rightarrow^{\cdot}\ r$
4. $[\odot]\omega_1 S : \pi[\odot]R :\rightarrow^{\cdot}\ l$
5. $\rightarrow^{\cdot}\ s$

which implements the following behaviour. ω_1 is stopped (s forever) unless π is in the same cell. In that case it moves in the opposite direction from where π came from.

The observation for Figure 7 for ω_1 is $: \pi\odot : L : \omega_1 S : \odot R ::$ and we apply the algorithm we have $: \pi\odot : L : \omega_1 S : \odot R :: s$ which is postprocessed leaving s.

For the complexity calculation, any approximation of the length of the algorithm (e.g. the number of symbols of the whole algorithm) would be valid as its complexity.

It is important to remark that the complexity of a set of objects is *not* the sum of the complexities of the objects. Again, the complexity of 100 exactly equal objects cannot be 100 times the complexity of one object, and should we smaller than the complexity of a fewer number of objects with more diversity. This is the spirit of Kolmogorov Complexity and we have to maintain it in the coding. An option is to generate a single Markov algorithm for all the objects, but in this case we go to the general solution at section 2 (although now with the assurance of good properties proven in the previous section). Alternatively, we suggest using an extra bit in each rule and then a way to refer to other rules in other objects. Consequently, if 100 objects share a lot of rules, the complexity of the whole set will be much smaller than the sum of the parts.

6.3 Generating the Environments and Using them in the Tests

Given the previous coding and generation guidelines, we have to deal with non-termination. In fact, this is not only a practical issue, but also a theoretical one, as we discuss in [6], since we do not want to have environments which take hours to interact. We want them to be almost immediate. So instead of Kolmogorov complexity, we use a computable (time-bounded) version, denoted by Kt^{max} (see [6] for details). This means that we do not need to compute the time complexity of all the Markov algorithms but just to set a bound for each execution. For the space generation we can be a little bit more flexible and leave more time, since their programs are computed only once. Once the space is generated, it is not modified. For the objects, the time bound must be a little bit tighter, in order to have (almost) instant interactions.

For the implementation of the test, a pool of spaces can be generated beforehand and also a set of objects which are compatible with each space. Then,

environments can be created using combinations from those pools. As we mentioned above, the coding (and hence complexity) of a set of objects is computed globally, by taking into account the common rules shared by several objects. This also suggests some heuristic ideas to ease the generation of this pool, such as evolutionary algorithms, where rules can mutate and be exchanged between objects, and also some wildcards for objects, instead of using the specific symbol for each in the rules.

7 Conclusions

Finding an environment class which is easy to generate, is universal, is unbiased and is useful for intelligence measurement (only discriminative environments are included and random agents are expected to have neutral rewards) is not easy, as we have realised. Some choices made in this paper can obviously be improved, and the same class or better classes might be more elegantly defined. However, to our knowledge, this is the first attempt in the direction of setting a general environment class for intelligence measurement which can be effectively generated and coded. It is interesting to compare this class, and especially the most simple environments generated by it, with some typical problems which are used to compare agent performance (see e.g. [33] or vmost especially [28], where we see 1D mazes, 2D mazes, door puzzles, 4x4 grids, tic-tac-toe, rock-paper-scissor game and a simplified pacman). We believe that our environment class is less biased than this selection.

The main idea for the definition of our environment class has been to separate the space from the objects, to be able to define a class which only includes observation and reward-sensitive environments which are balanced. The space sets some common rules on actions and the objects may include any universal behaviour. This opens the door to social environments. Two special symmetric objects are in charge of the rewards and some rules are defined on how these rewards are produced to make all the environments comply with the conditions, independently from their behaviour.

The next step is the implementation of intelligence tests based on this environment class and its application to biological subjects (human adults and children, chimpanzees and dolphins) and artificial ones (simple agents, reinforcement learning agents, agents based on approximations of Levin's optimal universal search [21] or on AIXI [15][16][28]). Analysing whether this environment class and the anytime intelligence test devised in [6] is able to practically measure the intelligence of all these systems is the ultimate goal of the project $anYnt$ (see http://users.dsic.upv.es/proy/anynt/) which clearly falls inside the general goal of Section 7 (Open Problems Defining Intelligence) in [17].

8 Acknowledgments

The author thanks the funding from the Spanish Ministerio de Educación y Ciencia (MEC) for projects EXPLORA-INGENIO TIN2009-06078-E, CONSOLIDER-

INGENIO 26706 and TIN 2007-68093-C02, and GVA project PROMETEO/2008/051. I am also grateful to David L. Dowe and Javier Insa for reading a draft of this paper and spotting several mistakes and typos.

References

1. Paul Almond. What is a low level language? 2005. http://www.paul-almond.com/WhatIsALowLevelLanguage.htm.
2. D. L. Dowe and A. R. Hajek. A non-behavioural, computational extension to the Turing Test. In *Intl. Conf. on Computational Intelligence & multimedia applications (ICCIMA'98), Gippsland, Australia*, pages 101–106, 1998.
3. D.L. Dowe and A.R. Hajek. A computational extension to the Turing test. In *Proceedings of the 4th Conference of the Australasian Cognitive Science Society, Newcastle, NSW*, 1997.
4. Dina Q. Goldin. Persistent Turing machines as a model of interactive computation. In Klaus-Dieter Schewe and Bernhard Thalheim, editors, *FoIKS*, volume 1762 of *Lecture Notes in Computer Science*, pages 116–135. Springer, 2000.
5. J. Hernández-Orallo. A (hopefully) non-biased universal environment class for measuring intelligence of biological and artificial systems. In M. Hutter et al., editor, *Artificial General Intelligence, 3rd Intl Conf*, pages 182–183. Atlantis Press, Extended report at http://users.dsic.upv.es/proy/anynt/unbiased.pdf, 2010.
6. J. Hernández-Orallo and D. L. Dowe. Measuring universal intelligence: Towards an anytime intelligence test. *Artificial Intelligence*, 174(18):1508 – 1539, 2010.
7. J. Hernández-Orallo and N. Minaya-Collado. A formal definition of intelligence based on an intensional variant of Kolmogorov complexity. In *Proceedings of the International Symposium of Engineering of Intelligent Systems (EIS'98)*, pages 146–163. ICSC Press, 1998.
8. José Hernández-Orallo. Beyond the Turing test. *Journal of Logic, Language and Information*, 9(4):447–466, 2000.
9. José Hernández-Orallo. Constructive reinforcement learning. *International Journal of Intelligent Systems*, 15(3):241–264, 2000.
10. José Hernández-Orallo. On the computational measurement of intelligence factors. In *Performance metrics for intelligent systems workshop*, pages 1–8. Gaithersburg, MD, 2000.
11. José Hernández-Orallo. Thesis: Computational measures of information gain and reinforcement in inference processes. *AI Communications*, 13(1):49–50, 2000.
12. José Hernández-Orallo. On discriminative environments, randomness, two-part compression and MML. *Technical Report, available at http://users.dsic.upv.es/proy/anynt/*, 2009.
13. E. Herrmann, J. Call, M.V. Hernández-Lloreda, B. Hare, and M. Tomasell. Humans have evolved specialized skills of social cognition: The cultural intelligence hypothesis. *Science*, 7 September 2007, Vol 317(5843):1360–1366, 2007.
14. Bill Hibbard. Bias and no free lunch in formal measures of intelligence. *Journal of Artificial General Intelligence*, 1(1):54–61, 2009.
15. Marcus Hutter. The fastest and shortest algorithm for all well-defined problems. *International Journal of Foundations of Computer Science*, 13:431–443, 2002.
16. Marcus Hutter. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*. Springer, 2005.

17. Marcus Hutter. Open problems in universal induction & intelligence. *CoRR*, abs/0907.0746, 2009. informal publication.
18. David Keil and Dina Q. Goldin. Indirect interaction in environments for multi-agent systems. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *E4MAS*, volume 3830 of *Lecture Notes in Computer Science*, pages 68–87. Springer, 2005.
19. Shane Legg. *Machine Super Intelligence*. Department of Informatics, University of Lugano, June 2008.
20. Shane Legg and Marcus Hutter. Universal intelligence: A definition of machine intelligence. *Minds and Machines*, 17(4):391–444, 2007. http://www.vetta.org/documents/UniversalIntelligence.pdf.
21. L. A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973.
22. Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications (3rd ed.)*. Springer-Verlag New York, Inc., 2008.
23. Andrey Andreevich Markov. The theory of algorithms. *American Mathematical Society Translations, series 2*, 15:1–14, 1960.
24. Markus Mueller. Stationary algorithmic probability. *CoRR*, abs/cs/0608095, 2006. http://arxiv.org/abs/cs/0608095.
25. Florentin Neumann, Andrea Reichenberger, and Martin Ziegler. Variations of the Turing test in the age of internet and virtual reality. In *Proceedings of the 32nd annual German conference on Advances in artificial intelligence*, pages 355–362, 2009. http://arxiv.org/PS_cache/arxiv/pdf/0904/0904.3612v1.pdf.
26. Barney Pell. A strategic metagame player for general chesslike games. In *AAAI*, pages 1378–1385, 1994.
27. J. Rissanen. A universal prior for integers and estimation by minimum description length. *Annals of Statistics*, 11(2):416–431, 1983.
28. Joel Veness, Kee Siong Ng, Marcus Hutter, and David Silver. A Monte Carlo AIXI approximation. *CoRR*, abs/0909.0801, 2009. informal publication.
29. C. S. Wallace. *Statistical and Inductive Inference by Minimum Message Length*. Ed. Springer-Verlag, 2005.
30. D.A. Washburn and R.S. Astur. Exploration of virtual mazes by rhesus monkeys (macaca mulatta). *Animal Cognition*, 6(3):161–168, 2003.
31. Danny Weyns, H.V.D. Parunak, F. Michel, Tom Holvoet, and J. Ferber. Environments for multi-agent systems, state-of-the-art and research challenges. In *Environments for multi-agent systems*, volume 3374 of *Lecture Notes in Computer Science*, pages 1–48. Held with the 3th Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS, Springer-Verlag, 2005.
32. Damien Woods and Turlough Neary. The complexity of small universal Turing machines. In S. Barry Cooper, Benedikt Lï¿½we, and Andrea Sorbi, editors, *CiE*, volume 4497 of *Lecture Notes in Computer Science*, pages 791–799. Springer, 2007.
33. Z. Zatuchna and A. Bagnall. Learning mazes with aliasing states: An lcs algorithm with associative perception. *Adaptive Behavior*, 17(1):28–57, 2009.