Rational OpenCog Controlled Agent

Nil Geisweiller and Hedra Yusuf

SingularityNET Foundation, The Netherlands
{nil,hedra}@singularitynet.io

Abstract. TODO

Keywords: Symbolic Reinforcement Learning · Pattern Mining · Temporal Reasoning · Thompson Sampling · OpenCog · Minecraft

1 Introduction

This paper describes an attempt to leverage the OpenCog framework [**?**] for controlling agents in uncertain environments. It can be seen as a reboot of previous attempts [**?**, **?**, **?**, **?**, **?**] relying on new or improved components such as

- a hypergraph pattern miner [**?**] and a version of PLN [**?**] both implemented on top of OpenCog's Unified Rule Engine equipped with an inference control mechanism;
- a temporal and procedural extension of PLN [**?**];
- a simplified version of OpenPsi [**?**, **?**] leaving aside built-in urges and modulators from MicroPsi [**?**] and using an action selection policy based on Thompson Sampling [**?**].

It is comparable to OpenNARS for Applications (ONA) [**?**] but, among other differences, uses PLN [**?**] as its core logic rather than NAL [**?**].

The ultimate goal of this project is to provide a technology to enable us to experiment with high forms of meta-learning and introspective reasoning for self-improvements and growth. The rational for using a reasoning-based system is that it offers maximum transparency and is thus more amenable to reflectivity and introspection [**?**, **?**]. The work that is described in this paper is only the premise of that goal. No meta-learning is taking place yet. The objective for now is to build an agent that is able to discover regularities from its environment and acts rationally, possibly at the expense of efficiency, at least initially. For discovering regularities, the agent uses a reasoning-based pattern miner [**?**]. Then combine these regularities to form plans using temporal and procedural reasoning. More specifically plans are predictive implications of the form

$$C \wedge A \rightsquigarrow^{T} G \stackrel{\text{m}}{=} TV$$

called *Cognitive Schematics* or *Schemata*. Which can be read as *"in some context* C*, if some, possibly composite, action* A *is executed, then after* T *time units, the*

goal G is likely to be fulfilled, with second order probability described by TV". These plans are then combined to into a mixture that grossly approximates Solomonoff distribution [**?**]. Finally, the next action is selected using Thompson Sampling [**?**]. The resulting software is called ROCCA for *Rational OpenCog Controlled Agent*.

The rest of the paper is organized as follows. A recall of the OpenCog framework is provided in Section 2. The ROCCA software is described in Section 3. An experiment using it to control an agent in Minecraft is described in Section 4. A conclusion including future work and directions is given in Section 5.

2 OpenCog Framework Recall

OpenCog [**?**] is a framework offering a hypergraph database technology with a query language and a collection of programs built on top of it to perform cognitive functions such as learning, reasoning, spreading attention and more. Knowledge is stored in *AtomSpaces*, hypergraphs composed of atoms, links and nodes, where links can connect to other atoms. Values can be attached to atoms to holding probability, confidence, importance and more. Values and atoms can be of various types. Let us recall the types we need for the rest of paper.

- A *TruthValue* is a second order probability distribution, i.e. the probability of a probability.
- A *SimpleTrueValue* is a particular TruthValue where the second order distribution is represented by a beta distribution of parameters controlled by a *strength*, a proxy for a probability, and a *confidence* over that probability. It is denoted $< s, c >$ where s is the strength and c is the confidence.
- A *Predicate* is function from a domain D to *Boolean*. A TruthValue can be attached to a predicate, representing the prevalence of its satisfied inputs. For instance $P \stackrel{\text{m}}{=} < 0.4, 0.1 >$ represents that P tends to evaluate to *True* 40% of the time, but there is a small confidence of 0.1 over that 40%. A TrueValue can be attached to individual evaluations as well. For instance $P(a) \stackrel{\text{m}}{=} < 0.9, 1 >$ represents that the probability of $P(a)$ evaluating over a particular a to *True*, is 0.9 and we are certain about it.
- A *Conjunction* is a link between two predicates, representing the predicate resulting from the pointwise conjunction of these two predicates. For instance $P \wedge Q \stackrel{\text{m}}{=} < 0.2, 0.3 >$ represents the prevalence, with strength 0.2 and confidence 0.3, of the pointwise conjunction of P and Q.
- An *Implication* is a link between two predicates, semantically representing the conditional probability between two events represented by these predicates. For instance $P \rightarrow Q \stackrel{\text{m}}{=} < 0.7, 0.4 >$ indicates that if $P(x)$ is *True* then there is a 70% change with a 0.4 confidence, that $Q(x)$ is also *True*.

Additionally we use the following types for temporal reasoning.

- A *Sequential Conjunction* is a link between two temporal predicates, representing the predicate resulting from the pointwise conjunction of these

predicates while the second one leads by a certain time. For instance $P \curlywedge^T Q$ is the pointwise conjunction of P and a leading Q by T time units. Meaning that for $(P \curlywedge^T Q)(x,t)$ to be *True*, requires that $P(x,t)$ and $Q(x,t+T)$ be *True* as well.

– A *Predictive Implication* is a link between two temporal predicates, representing the conditional probability between two events lagging by a certain time. For instance $P \rightsquigarrow^T Q \stackrel{\text{m}}{=} <0.8, 0.5>$ indicates that if $P(x)$ is *True* then there is a 80% chance with a 0.5 confidence, that after T time units $Q(x)$ will also be *True*.

The difference between a temporal and an atemporal predicate is its domain. The domain of a temporal predicate must have a dimension respecting certain properties such as having a total order. More detail about the temporal types and their associated inference rules is provided in [?].

3 Rational OpenCog Controlled Agent

ROCCA is implemented as an observation-planning-action loop interleaved with learning and reasoning. It provides an interfacing between OpenCog and environments such as Malmo [?] or OpenAI Gym [?]. It is written in Python which is both supported by these environments and OpenCog. Figure 1 provide a graphical representation of ROCCA as if it was a single loop incorporating all steps.

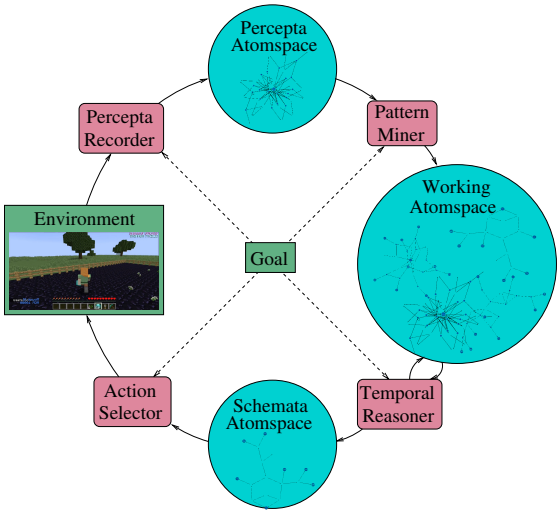

Fig. 1. ROCCA control and learning loops merged into one.

3.1 Memory

For better efficiency and clarity, the memory of the agent is split into three.

1. The *Percepta* [1] AtomSpace holds timestamped observations as they come into the system.
2. The *Working* AtomSpace holds any kind of data, ranging from timestamped observations to predictive implications. Most knowledge used and inferred during the course of learning are usually dumped into this AtomSpace.
3. The *Schemata* AtomSpace holds *Cognitive Schematics*, which are predictive implications relating contexts, actions and goals.

3.2 Processes

The agent runs two main processes:

1. *Control* for real-time reactive agent control.
2. *Learning* for non-reactive background learning.

In principle these two processes should happen in parallel. For now they alternate in series. The agent starts in a control phase. A number of control cycles occur as the agent motor-babbles through its environment. It then follows by a learning phase when the agent discover regularities and build plans. And finally repeats the control phase to test how it performs after learning.

3.3 Control

The control process is composed of control cycles, each decomposed into *Observation*, *Planning* and *Action* phases, as described below.

1. *Observation*:
   (a) Receive and timestamp observations, reward included, from the environment.
   (b) Store the timestamped observations in the Percepta AtomSpace.
2. *Planning*:
   (a) Select the goal for that cycle.
   (b) Find plans fulfilling that goal from the Schemata AtomSpace.
   (c) Build a mixture distribution from these plans.
3. *Action*:
   (a) Select the next action via Thompson Sampling according to that mixture distribution.
   (b) Timestamp and store the selected action in the Percepta AtomSpace.
   (c) Run the selected action and by that update the environment.

[1] Percepta means percepts in Latin. It is the plurial form of perceptum. Latin was chosen over English so that the difference between singular and plurial does not reduce to a single letter which can be prone to error when reading and writing code.

None of these steps are computationally expensive. They involve algorithms that are at most linear with the size of the Percepta and Schemata AtomSpaces. As time goes and knowledge accumulates however, it will progressively slow down. Indeed, for real-time responsiveness such control cycle must be bound by a constant. Achieving this may require to incorporate other mechanisms such as filtering and forgetting. This problem, as important as it is, for now is left for future research. Given the small environments ROCCA has been tested with, it has not been a problem so far. Let us now provide more details about these three phases.

Observation During the observation phase, data coming from the environment are timestamped and stored in the Percepta AtomSpace. The format used to represent that is *datum@timestamp*. For instance if at cycle 3 the agent observes *outside(house)* and *hold(key)*, then *outside(house)@3* and *hold(key)@3* are inserted into the Percepta AtomSpace.

Planning The first step of the planning phase is to select a goal G to fulfill. That is an important step that has been elaborated in [**?**, **?**]. In the current version of ROCCA though it merely returns a constant goal which is to gain a reward within a forward window. Once a goal has been selected, the agent search the Schemata AtomSpace with the following pattern matcher query

$$\$C \wedge \$A \rightsquigarrow^{T} G$$

where

- $\$C$ is a variable representing the context,
- $\$A$ is a variable representing the action or action plan [2],
- T is a time delta selected within that forward window,
- G is the selected goal.

All returned candidates are then filtered according to their contexts, only retaining those for which the context is evaluated to *True* at the current time. Ideally, such crisp evaluation should be replaced by a second order probability evaluation of a context being *True*. This is important for contexts that have elements of uncertainty. But for the sake of simplicity, in our experiments so far, all contexts are crisply evaluated. Then from the set of valid cognitive schematics, a second order mixture distribution is built and handed to the next phase for performing action selection. The calculations used to build that second order mixture distribution is detailed in [**?**].

Action The Action phase consists of the following steps:

[2] Variables are actually typed so that the pattern matcher cannot confuse what is context and action.

1. Select the next action via Thompson Sampling [**?**] according to the mixture distribution built during the planning phase.
2. Timestamp and store the selected action in the Percepta AtomSpace.
3. Run the selected action and updates the environment.

The trickiest step here is selecting the next action via Thompson Sampling. The novelty is that the second order probabilities can be leveraged by Thompson Sampling. For example, assume we have two actions, A_1 and A_2, to choose among two predictive implications

$$C_1 \wedge A_1 \rightsquigarrow^T G \stackrel{\text{m}}{=} <0.6, 0.9>$$
$$C_2 \wedge A_2 \rightsquigarrow^T G \stackrel{\text{m}}{=} <0.7, 0.1>$$

Using only the strengths of the truth values as proxy for probability of success, the choice is clear. Action A_2 should be selected, because its probability of success, which is 0.7, is greater than that of A_1, which is 0.6. However once introducing confidence, that choice becomes less clear because the truth value of success of A_2 has a low confidence of 0.1. In that case, first order probabilities are sampled from their corresponding second order distributions, and then these probabilities are compared. The action with the maximum probability gets selected. Informally, the idea is to consider the possibilities that the agent might be living in a world where A_2 has a lower probability of success than A_1. That is the essence of Thompson Sampling. Figure 2 shows the second order distributions of

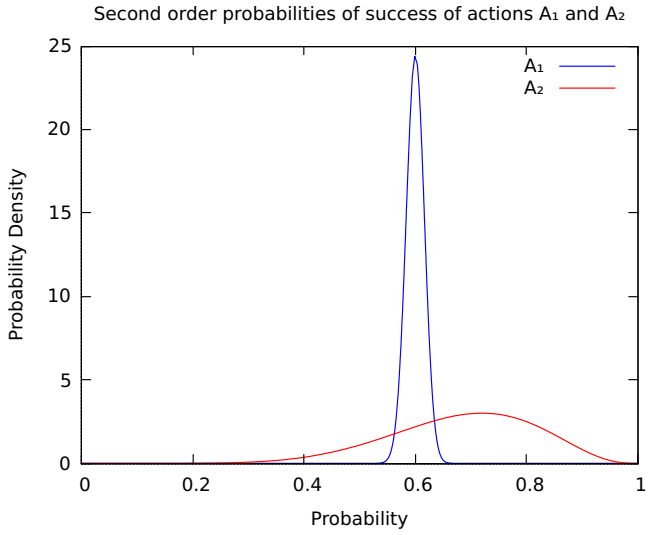

Fig. 2. Second order probability distributions of success of actions A_1 and A_2, using as parameters of the beta distribution $\alpha(s,c) = \alpha_0 + \frac{s.c.k}{1-c}$ and $\beta(s,c) = \beta_0 + \frac{(1-s).c.k}{1-c}$ where k, the *lookahead*, is set to 100, and α_0 and β_0 are set to Jeffreys prior.

the probabilities of success of A_1, in blue, and A_2, in red, for these truth values. As one may notice, the area under the red curve situated at the left of the blue curve is non-negligible. Meaning that the probability of being in a world where A_1 has a higher probability of success than A_2 is non-negligible as well. Because these strengths and confidences are periodically updated during the lifetime of the agent, one can see how Thompson Sampling is a great alternative to ε-greedy, as it offers a parameter-free mechanism to balance exploration and exploitation that dynamically adapts with the knowledge of the agent.

Note that in this example only two actions among two cognitive schematics are considered, but in practice there is usually a handful of actions among a potentially very large number of cognitive schematics with overlapping contexts and conflicting goals. The resulting distribution of success of each action is typically multi-modal and do not reduce to a beta distribution. How to deal with such a multitude of cognitive schematics is treated in [**?**].

3.4 Learning

The difficulty then comes down to discovering cognitive schematics that are as predictive and widely applicable as possible. For that ROCCA uses a combination of pattern mining and reasoning.

Pattern Mining A relatively inexpensive way to discover regularities in the environment is to mine the Percepta AtomSpace. For instance, given the following observation records

$$\{go(right)@0,\ square(right)@1,\ go(left)@1,\ square(left)@2, \ldots\}$$

the pattern miner can discover temporal relationships such as

$$go(right) \rightsquigarrow^1 square(right)$$
$$go(left) \rightsquigarrow^1 square(left)$$

as well as more abstract forms such as

$$go(x) \rightsquigarrow^1 square(x)$$

The pattern mining algorithm used by ROCCA is detailed in [**?**]. This is a generic hypergraph pattern miner, not specialized for temporal patterns. In order to mine temporal patterns with it, timestamps are represented as naturals. 0 is presented by Z, 1 by $S(Z)$, 2 by $S(S(Z))$, etc. This provides the needed structure to discover temporal relationships between events. As it currently stands, the pattern miner can efficiently discover mono-action plans. Mining poly-action plans is also possible but has two problems:

1. In the worse case, the computational cost of mining goes up exponentially with the size of the action sequence to mine.

2. The number of observations to accumulate in order to generate cognitive schematics with decent confidences goes up exponentially as well.

The latter is really the most problematic because we cannot buy our way out of it. If each observation takes a certain amount time, determined by the control cycle period in the case of primary observations, then we have to go through them, we cannot speed time up. This is even more true for abstract percepta that can only be observed at periods that are multiples of control cycle periods. Also, in some cases, a particular action sequence may never be observed at all, yet we still would like to have a way to estimate the likelihood of its success. In order to address these limitations and more, we need reasoning.

Temporal Reasoning ROCCA uses a temporal extension of PLN described in [**?**] to update existing cognitive schematics obtained by pattern mining, and discover new cognitive schematics by combining existing ones. For instance it can infer poly-action plans by stringing mono-action plans together, as well as generalize or specialize their contexts or goals. Temporal rules integrated into ROCCA include:

1. *Predictive Implication Direct Introduction* to infer the truth value of a predictive implication from direct observations.
2. *Temporal Conditional Conjunction Introduction* to specialize a goal within a plan by considering the conjunction of existing cognitive schematics goals.
3. *Temporal Deduction* to string together plans to form bigger ones.

The precise semantics of these rules is detailed in [**?**]. An example of how they are used is presented in Section 4.

4 Experiment with Simple Minecraft Environment

In this experiment, we utilized Malmo [15] to construct a basic Minecraft world that comprises a house filled with diamonds and a key. The objective of the agent is to locate the key, which is placed somewhere in the proximity of the house, retrieve it, and then unlock the door of the house. Upon unlocking the door, the agent is able to collect the diamonds and receive a reward.

The aim of the experiment is to make the ROCCA agent learn from its actions and perceptions in the Minecraft environment and do efficient planning so as to be able to collect as many diamonds as possible and accumulate reward. The ROCCA agent performs a series of possible actions with a goal of achieving a reward and learns from them by applying PLN and Pattern Miner. For simplicity purposes, the list of actions provided by the Minecraft has been replaced by an abstract action and perception where unnecessary details have been omitted. With that, we generate abstract actions such as go(key), go(house) and go(diamonds) where each of them contains a series of actions, and an abstract perception about the agent such as outside(house), hold(key) as well as the reward of completing a given action, reward(1) etc.

We conducted various experiments by adjusting different parameters. As demonstrated in our demo video (available at https://youtu.be/rCvQHrJAD2c), the ROCCA agent successfully learned various cognitive schematics leading to a goal. The experiment consisted of two iterations of the learning-training process, each lasting fifty iterations. During the first cycle, no learning took place as the agent had no prior knowledge about its environment. Therefore, the agent explored the environment and built its knowledge base by performing a combination of fifty randomly weighted actions. In the second cycle, the agent had enough background knowledge to apply reasoning as described in section 3.3, and discovered desired cognitive schematics which ultimately led to achieving the goal of receiving a reward.

Experimental result Let's look into how the ROCCA agent discovers temporal relationships from time stamped actions and perceptions. For instance, given the following observation during the second learning-training cycle

$$\{\textit{Reward(0)@50},\ \textit{outside(house)@50},\ \textit{hold(key)@50},\ \textit{go(house)@50},\ \textit{inside(house)@51},\ \textit{go(diamond)@51},\ \textit{Reward(0)@51},\ \textit{Reward(1)@52}, \ldots\}$$

The ROCCA agent acquires cognitive schematics through the Pattern Miner and infer their truth value using the Predictive Implication rule.

$$\textit{hold(key)} \wedge \textit{go(house)} \rightsquigarrow^{1} \textit{inside(house)} \stackrel{\text{m}}{=} <0.833, 0.007>$$

Additionally, by applying Temporal Conditional Conjunction Introduction rule on the relevant relationships, such as

$$\textit{nextto(closed\_door)} \wedge \textit{go(key)} \rightsquigarrow^{1} \textit{outside(house)} \stackrel{\text{m}}{=} <1, 0.007>$$
$$\textit{nextto(closed\_door)} \wedge \textit{go(key)} \rightsquigarrow^{1} \textit{hold(key)} \stackrel{\text{m}}{=} <1, 0.007>$$

the agent derives inputs for the Temporal Deduction rule,

$$\textit{hold(key)} \wedge \textit{outside(house)} \wedge \textit{go(house)} \rightsquigarrow^{1} \textit{inside(house)} \stackrel{\text{m}}{=} <0.833, 0.007>$$
$$\textit{Reward(0)} \wedge \textit{inside(house)} \wedge \textit{go(diamond)} \rightsquigarrow^{1} \textit{Reward(1)} \stackrel{\text{m}}{=} <1, 0.006>$$

ultimately resulting in the discovery of an effective plan that leads to achieving a positive goal.

$$\textit{hold(key)} \wedge \textit{outside(house)} \wedge \textit{go(house)} \nearrow^{T} \textit{go(diamond)} \rightsquigarrow^{T+1} \textit{Reward(1)} \stackrel{\text{m}}{=} <0.833, 0.005>$$

In this experiment, we evaluate the agent's performance based on two factors: the cognitive schematics it has learned and the rewards it has accumulated. Our results indicate that the ROCAA agent successfully learned the necessary cognitive schematics, resulting in a higher reward collection in the second cycle. However, these findings only apply to a simple Minecraft environment with a limited number of actions and may not be indicative of the overall performance of the ROCAA agent. Therefore, further extensive experiments are required to draw definitive conclusions about its performance, which will be pursued in future work.

5 Conclusion

ROCCA, a system that leverages the OpenCog framework for controlling an agent in uncertain environments has been presented. This agent is in a strong sense fully reasoning-based, from learning to planning. The advantage we believe of such approach, in spite of its current inefficiencies, is to offer greater transparency and enable greater capabilities for meta-learning and self-growth. As such, we are only at the start of our endeavor. Towards that end, future directions include:

1. Integrate *Economic Attention Networks* [**?**] for *Attention Allocation*. Record attentional spreading as percepta to learn Hebbian links [**?**] and improve attention allocation in return.
2. Carry out concept creation and schematization, also called crystallized attention allocation, to speed up repetitive information processing. This done well should also provide a solution to the problem of creating hierarchies of abstract observations and actions.
3. Record more internal processes, not just attentional spreading, as internal percepta to enable deeper forms of introspection.
4. Plan with internal actions, not just external, such as parameter fine-tuning and code rewriting to enable self-improvements.