
# Università degli Studi di Padova

Department of Information Engineering


**MASTER THESIS IN COMPUTER ENGINEERING**

# A new scope for Open Cognition project: task and motion planning through its Neural-Symbolic Knowledge Store


*Supervisor*
EMANUELE MENEGATTI
Università di Padova

*Co-supervisor*
ENRICO PAGELLO
ELISA TOSELLO
Università di Padova

*Master Candidate*
Michele THIELLA


Date
18 October 2021

*Academic Year 2020 - 2021*

# Abstract

# Contents

# List of Figures

# List of Tables

Road to agi.

# 1 Introduction

As industrial robots become faster, smarter, and cheaper, more and more companies are beginning to integrate this technology in conjunction with their workforce. Nowadays, robots have a range of applications that cover many different areas. These applications are no longer limited in structured environments, where the robot behavior could be directly specified by a human.

Where early robots blindly followed the same path, and later iterations used lasers or vision systems to detect the orientation of parts and materials, the latest generations of robots can integrate information from multiple sensors and adapt their movements in real time. They can also make use of more powerful computer technology and big data–style analysis.

Advances in artificial intelligence and sensor technologies allow robots to cope with a far greater degree of task-to-task variability. However, their ability to adapt to different tasks is still a long way from a general level of adaptability.

Until now, for example, every industrial robot is designed with a specific purpose. Therefore, a new robot is often needed to perform a new task. Even the most advanced robots, which exploit Neural Networks (NN) and Artificial Intelligence (AI), are unable to easily learn a new task.

It is necessary to design the AI architecture carefully and, to be successful, people must stop misusing the term AI. This term is used by generalizing those that are the three types of AI: Narrow-AI, General-AI, and Super-AI. The distinction between them is the key to technological progress, especially in the robotic field.

## 1.1 Artificial Intelligence

The following is a brief description of these three AI types, the Articles [1, 2, 3, 4, 5, 6] talk more about this.

1. Narrow-AI:
   Current AI technologies all fall under the Artificial Narrow Intelligence (ANI or Narrow-AI) category, which means they are very good at only one or a few closely related tasks. This type of AI has a limited range of abilities, specifically designed for a narrow use. It is able to reach a level of performance of a human, and even better, but only within this limited field that is its specialty. Examples of ANI include everything from Siri, Face ID and the Google Assistant, to self-driving cars and DeepMind's board game playing program. This is the only form of AI that has been developed so far.

2. General-AI:
   The next step after ANI is Artificial General Intelligence (AGI or General-AI), much more similar to human intelligence and not focused on specific tasks. It would be similar to a human mind and in theory, it should be able to think and function like it, being able to make sense of different content, understand issues and decide what is best in a complex situation. AGI hasn't been achieved yet. There isn't the technical capacity of producing something as complex yet, and there is also no certain knowledge of how the human brain actually works either. AGI is a relatively logical and rational future though, and it could be attained at some point if humans develop their knowledge and understanding, as well as technical skills to a high enough level.
   An overview of AGI, including important reflections, written by Ben Goertzel, can be found here: [7].

3. Super-AI:
   When AGI is achieved and computers are able to learn independently at a very quick rate, and exponentially improve on their own without human intervention or help, the final step that AI could hypothetically reach is Artificial Super Intelligence (ASI or Super-AI). At this stage AI would be capable of vastly outperforming the best human brains in practically every field. The evolution from AGI to ASI would in theory be fast, since AGI would allow computers to "think" and exponentially improve themselves once they are able to really learn from experience and by trial and error.

The most important differences, easily understood from the definitions above, are between ANI and AGI, partly because ASI will only be considered when AGI exists. The timing forecasts and the difficulties of that step can be found in [8].
These differences immediately lead into much wider contexts than industry, as AGI seems

focused on complex environments such as real-world and human-computer interaction. However, using an AGI system in an industrial setting can have many benefits:

- Having a general knowledge base for the robot, which can be shared with other ones.
- Use the robot(s) to cover multiple and more complex tasks.
- Make cooperation with humans natural and encourage learning from them.
- Robot(s) is robust to changes in the environment, its state and its task.
- Easier implementation of new modules and their merge into the system.
- Not only maintain, but rather exploit existing Narrow-AI systems as modules of the AGI system, in order to take advantage of their potential and allow interaction between them in a "single" large knowledge base.

For this project, one of the proto-AGI[1] systems currently under research is used, which proposes a concrete system that demonstrates the feasibility of the concepts just listed, and more.

There are several projects and researches that want to achieve the AGI goal. One of these is the Elon Musk and Sam Altman's OpenAI Project. Its mission is to ensure that AGI, intended as highly autonomous systems that outperform humans at most economically valuable work, benefits all of humanity.
OpenAI is focused on Deep Neural Networks for almost all projects such as OpenAI Codex, its AI system that translates natural language to code [9], CLIP (Contrastive Language-Image Pre-Training) a general-purpose vision system, which is a Neural Network trained on a variety of image-text pairs [10] and one of the most important: Generative Pre-trained Transformer 3 (GPT-3). This is an autoregressive language model that uses Deep Learning to produce human-like text [11]. With over 175 billion parameters, it is the largest Neural Network ever created [12].
Although OpenAI is well-funded and achieving excellent results, there is a second AGI-oriented project that is noteworthy: the Open Cognition (OpenCog) project. It is also the system on which this project is based.

[1] Any current theoretical or practical concept about AGI falls into a category that is called proto-AGI for now. However, in this paper the two terms are considered interchangeable.

The reason for this choice, the preference of OpenCog over OpenAI, lies in the approach to solving the AGI problem.
As mentioned above, OpenAI appears to be based on the general plan of starting from current Deep Neural Network tech, applying and extending it in various interesting and valuable ways, and in this way moving incrementally toward AGI without that much of a general plan or model of the whole AGI problem.
On the other hand, OpenCog is founded based on a comprehensive model of human-like general intelligence, and a comprehensive overall plan for getting from here to human-level AGI. Thus, it has an integrative approach in which multiple different sorts of AI algorithms (Deep Neural Networks, Probabilistic Logic Theorem Proving, Evolutionary Learning, Concept Blending, etc.) operate together on a common representational substrate.
It is not about building more accurate classification algorithms, or more efficient computer vision systems, or better language processing or boutique information retrieval algorithms, diagnosing diseases, answering trivial questions or driving a car, etc. It is concerned with generic intelligence and the inter-related cognitive processes it entails. It is about making software that perform specific tasks, using structures and processes that appear capable of being extended to more and more general tasks.
The OpenCog system is explained in detail later, in the Chapter **2**.

OpenAI and OpenCog are two distinct ways, both valid. However, we believe that the AGI architecture shouldn't be based on Neural Networks, for several reasons (in addition to the general problems as in [13]):

- A huge amount of data is required.
  Figure 1.1b shows annual data growth versus growth in CPU processing power. The gap leads one to think that as the data required for NNs increases, hardware technology must be improved.

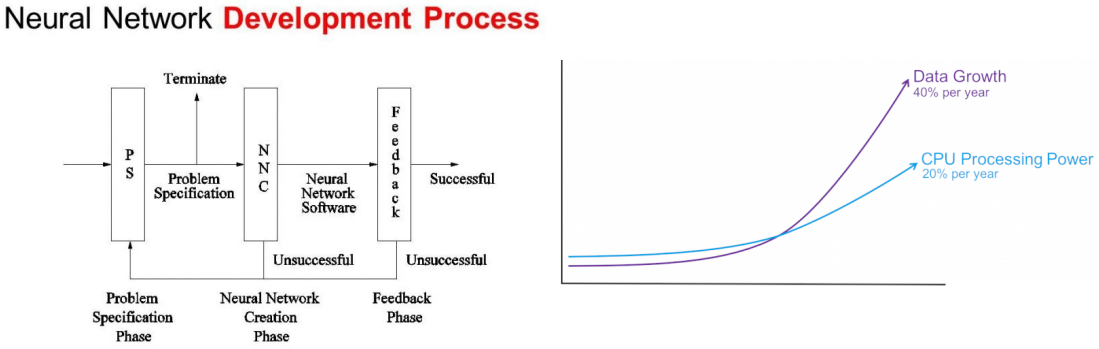
- Adversarial Examples problem: it is a way to deceive almost all AI classifiers easily.
  Figure 1.1c is an illustration of machine learning adversarial examples. Studies have shown that by adding an imperceptibly small, but carefully designed perturbation, an attack can successfully lead the machine learning model to making a wrong prediction. For more information see [14, 15, 16, 17].

- The cost: training GPT-3, for example, would have an estimated cost between $4.6 and $12 million [18, 19].
  Figure 1.1a gives a general idea of a NN development process .

- Neural Networks as black box: the best-known disadvantage of Neural Networks is their "black box" nature. It means that, while it can approximate any function, studying its structure won't give any insights on the structure of the function being approximated. More details in [20].

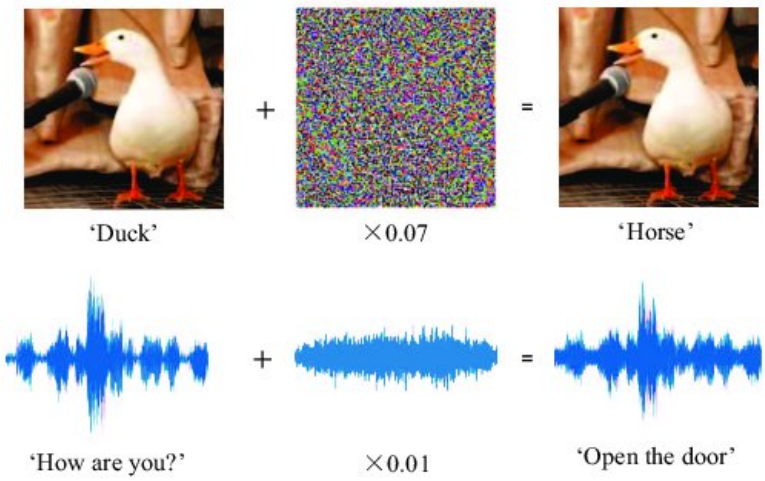
The disadvantages of an NN-based approach like OpenAI have been mentioned. Now, to understand the advantages of a human-like approach like OpenCog, it is necessary to explain the OpenCog system. Therefore, these advantages are postponed to the Section [sec:opencog advantages ...].

**Update:**




**(a)** Neural Network development process.
Image source: [21].

**(b)** Neural Network data process.
Image source: [22].




**(c)** Adversarial Examples.
Image source: [23].

**Figure 1.1:** Some disadvantages of Neural Networks.

# 2

# OpenCog System

The OpenCog design aims to capture the spirit of the architecture and dynamics of the brain without imitating the details (which are largely unknown) via:

- Integrating together a carefully selected combination of cognitive algorithms acting on different kinds of knowledge.
- A scalable, robust and flexible C++ software architecture.
- A manner specifically designed:
  - To cooperate together with "cognitive synergy" for the scope of tasks characteristic of human intelligence.
  - To give rise to the emergence of an effectively functioning knowledge network in the AI system's mind, as it interacts with the world, including a self-updating hierarchical/heterarchical ontology and models of itself and others.

Following section, Section 2.1, elaborates on the new concepts introduced in these points.

## 2.1 Cognitive Synergy

OpenCog is a diverse assemblage of cognitive algorithms, each embodying its own innovations. The power of the overall architecture is its careful adherence to the principle of Cognitive Synergy.

The human brain consists of a host of subsystems that perform particular tasks, both specialized and general in nature, connected together in a manner enabling them to synergetically assist, rather than work against each other.
The essential principles of Cognitive Synergy Theory (CST) can be summarized in the following points, further explored in [24]:

1. Intelligence can be understood as the ability to achieve complex goals in a certain set of environments.
2. An intelligent system requires a "multi-memory" architecture, meaning the possession of a number of specialized yet interconnected knowledge types.
3. "Cognitive processes": a system must possess knowledge creation mechanisms corresponding to each of these memory types.
4. Each cognitive process must have the ability to recognize when it lacks information and thus, draw it from knowledge creation mechanisms related to other types of knowledge.
5. The Cognitive Synergy is, therefore, represented by the interaction between the knowledge creation mechanisms, which perform much more effectively in combination than non-interactive mode.
6. The activity of the different cognitive processes involved in an intelligent system can be modeled in terms of the schematic implication "Context & Procedure $\rightarrow$ Goal".

These points are implicit in the systems theory of mind given in [25], where more thorough characterizations of these ideas can be found.
Interactions as mentioned in Points 4 and 5 are the conceptual core of CST.
Most AI algorithms suffer from combinatorial explosions. In a "general intelligence" context, there is a lack of intrinsic constraint; consequently, the algorithms are unable to filter through all the possibilities (as opposed to a ANI problem like chessplaying, where the context is huge but constrained and hence restricts the scope of possible combinations that needs to be considered).
To decrease the severity of combinatorial explosions, one can use an AGI architecture based on CST, in which the different learning mechanisms dealing with a certain sort of knowledge, are designed to synergize with ones dealing with other sorts of knowledge.
It is necessary that each learning mechanism recognizes when it is "blocked" and then, it can ask for help to the other complementary cognitive mechanisms.

The Figure 2.1 is proposed to give a general visual idea of these concepts. It shows an overview of the most important cognitive dynamics considered in Cognitive Synergy Theory and describes the behavior of a system as it pursues a set of goals, which are then refined by inference (through a logic engine or as an emergent process resulting from the dynamics of an Neural Network system), aided by other processes.

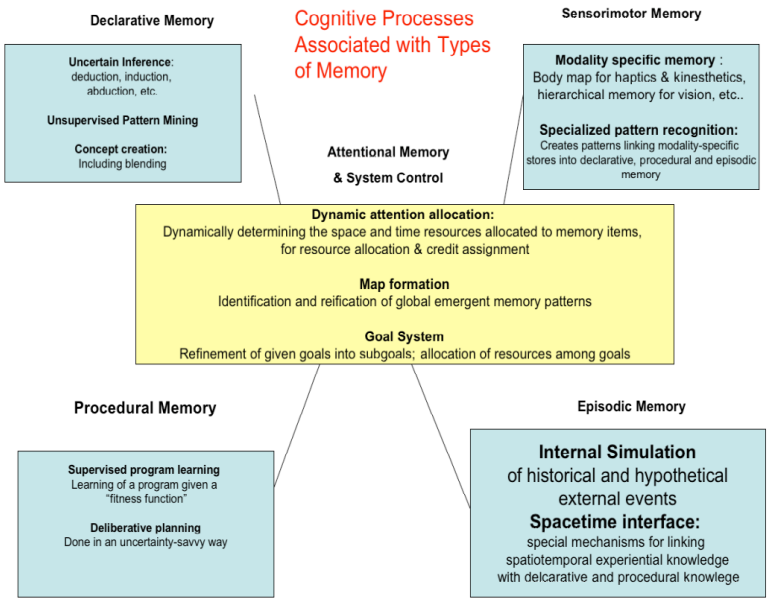



**Figure 2.1:** A high-level overview of the main types of cognitive process considered in Cognitive Synergy Theory, categorized according to the type of knowledge with which each process deals.

The detailed argument explaining how the cognitive algorithm selection and integration methods, chosen by the OpenCog team, will have the desired effect, is sketched on the OpenCog wiki site[1] and various previously-published conference papers. It has been presented more thoroughly in the 2014 books Engineering General Intelligence vol. 1 and 2 [26, 27].

## 2.2 OpenCog Architecture

There are several components that make up the basic architecture of OpenCog. In the following sections, they will be described, more or less independently, before being concatenated and contextualized into the problem considered in this project.

[1] https://wiki.opencog.org/w/Background_Publications

Currently, the OpenCog project is under strong development and some of the concepts presented here may be obsolete, improved or evolved, or being redesigned. However, much of the basic infrastructure and theory remains unchanged.

### 2.2.1 Atomspace

The AtomSpace is a platform for building Artificial General Intelligence (AGI) systems. It provides the central knowledge representation component for OpenCog. As such, it is a fairly mature component, on which a lot of other systems are built, and which depend on it for stable, correct operation in a day-to-day production environment.
It is a mashup of a large variety of concepts from mathematical logic, theorem proving, graph theory, database theory, type theory, model theory and knowledge representation.
More specifically, the OpenCog AtomSpace is an in-RAM knowledge representation database, an associated query engine and graph-re-writing system, and a rule-driven inferencing engine that can apply and manipulate sequences of rules to perform reasoning.
The best way to capture all of this, is a kind of in-RAM Generalized Hypergraph (Metagraph) Database.
On top of this, the Atomspace provides a variety of advanced features not available anywhere else. It is currently used to store natural language grammars, dictionaries and parsers, to store biochemical and biomedical data, robot control algorithms, machine learning algorithms, audio/video processing pipelines and deep learning neural networks.

#### 2.2.1.1 Metagraph: a Generalized Hypergraph

Formally, a graph is:

- A set of vertexes $V = \{v_1, v_2, \cdots, v_M\}$
- A set of edges $E = \{e_1, e_2, \cdots, e_N\}$ where each edge $e_k$ is an ordered pair of vertexes drawn from the set $V$.

Since edges are ordered pairs, it is conventional to denote them with arrows. In practice, one wishes to associate a label to each vertex, and also some additional attribute data (e.g. weight); likewise for the edges.

A hypergraph is very similar to a graph, except for the edges. "Hyperedges" are defined as edges that can contain more than two vertices. That is, the hyperedge, rather than being an ordered pair of vertices, is an ordered list of vertices.
Formally, a hypergraph is:

- A set of vertexes $V = \{v_1, v_2, \cdots, v_M\}$
- A set of hyperedges $E = \{e_1, e_2, \cdots, e_N\}$ where each hyperedge $e_k$ is an ordered list of vertexes drawn from the set $V$. This list may be empty, or have one, or two, or more members.

A good representation of a hypergraph is the one proposed in Figure 2.2. It was a straightforward extension of the edge table, to which a new column was added for each position and then, those columns were mashed together into one set.

| vertex id | incoming-set | attr-data |
|---|---|---|
| $v_1$ | $\{e_1, e_2, e_4\}$ | ... |
| $v_2$ | $\{e_2, e_4\}$ | ... |
| $v_3$ | $\{e_3, e_4\}$ | ... |
| $v_4$ | $\{e_2\}$ | |

**Figure 2.2:** Hypergraph edge table

The vertex table looks like the edge table, but the vertex-list is an ordered list, while the incoming-set (the edge-set) really is a set. This is because a hypergraph is "almost" a bipartite graph, having the form of Figure 2.3, with the set E on the left being the set of hyperedges.

Before define a Metagraph, a change of terminology is useful: the basic objects are now called "Nodes" and "Links" instead of "vertexes" and "edges".
Thus, a Metagraph is:

- A set of nodes $V = \{v_1, v_2, \cdots, v_M\}$
- A set of links $E = \{e_1, e_2, \cdots, e_N\}$ where each hyperedge $e_k$ is an ordered list of nodes, or other links, or a mixture. They are arranged to be acyclic (to form a directed acyclic graph).

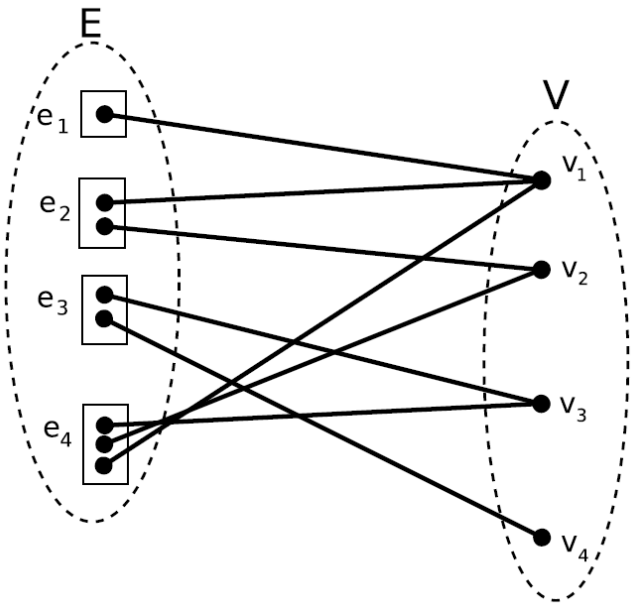



**Figure 2.3:** The $E$ and $V$ ellipses are the hyperedge and vertex tables. The boxes mean that the hyperedges are ordered lists.

The metagraph is a generalization of a hypergraph, in the sense that now a hyperedge (link) may contain either another vertex (node) or another link. Visually, it has the shape of a Directed Acyclic Graph (DAG), such as the one shown in Figure 2.4.

But in Metagraph, links are ordered lists, represented as boxes. Thus, to convert it in a DAG it is possible to collapse the boxes to single points or to dissolve the boxes entirely and replace a single arrow, from point-to-box, by many arrows, from point to each of the box elements.
Whereas the node table is the same as the vertex table for the hypergraph, nevertheless the link table now requires both an outgoing-atom list and an incoming-link set. Figure 2.5 shows the result.

For convenience, the name "Atom" is given to something that is either a node or a link. Links are then, sets of atoms.
These concepts are described in [28, 29], where RAM-usage considerations, reasons why metagraphs offer more efficient, more flexible and more powerful ways of representing graphs and reasons why a metagraph store is better than a graph store and much more, can also be found.

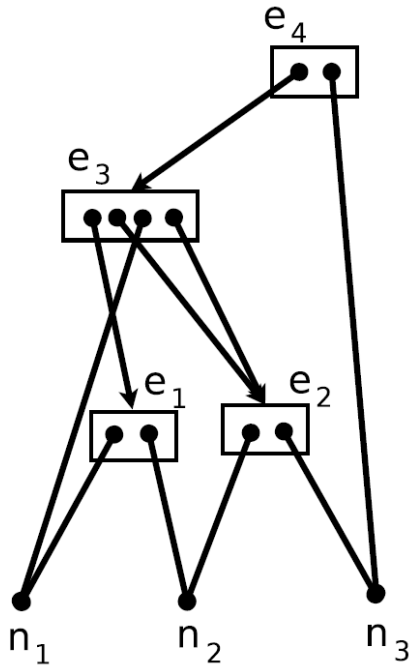



**Figure 2.4:** An example of a DAG.

Finally, some extensions of the metagraph are considered: Typed Metagraph (TMG) and Directed Typed Metagraph (DTMG).
Typed metagraphs are defined as hypergraphs with types assigned to hyperedges and their targets, and the potential to have targets of hyperedges connect to whole links, as well as targets. An example can be found in Figure 2.6.

A natural extension of a TMG is the Probabilistically TMG, based on probabilistic dependent types. Thus, one can assign a probability (or an entire probability distribution) to each connection between edges, thanks to the probabilistic type inheritance relations. In this way, it is possible to obtain a KB that can work with probabilistic logic, fuzzy logic, make uncertain inferences and much more.
Atomic Directed Typed Metagraphs (atomic DTMG) are introduced via partitioning the targets of each edge in a typed metagraph into input, output and lateral sets; one can then look at "metapaths" in which edges' output-sets are linked to other edges' input-sets (Figure 2.7). Thus, a DTMG is generally defined as a TMG composed by connecting DTMGs via metapaths, a recursive definition that bottoms out on the definition of atomic DTMGs.
For the whole theoretical formalism concerning TMG and DTMG refer to [30]. That pa-

| link id | outgoing-list | incoming-set | attr-data |
|---|---|---|---|
| $e_1$ | $(n_1, n_2)$ | $\{e_3\}$ | ... |
| $e_2$ | $(n_1, n_3)$ | $\{e_3\}$ | ... |
| $e_3$ | $(e_1, e_2, n_1, e_2)$ | $\{e_4\}$ | ... |
| $e_4$ | $(e_3, n_3)$ | $\{\cdot\}$ | |

**Figure 2.5:** Metagraph link table

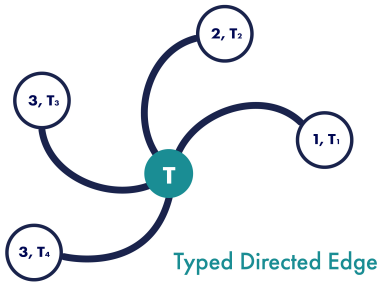



**Figure 2.6:** Example of a typed, directed edge. This one has 4 targets. $T$ is the type of the edge itself. The third and fourth targets are unordered relative to each other.

per concludes by also describing useful types of morphisms that can be defined on a DTMG (catamorphisms, anamorphisms, histomorphisms, futumorphisms, hylomorphisms, chronomorphisms, metamorphisms and metachronomorphisms). They allow to formulate a wide variety of operations on metagraphs, which will not be described here.
The important thing to keep in mind is that the KB is used for AGI, so there will be many metagraphs and very large ones. Morphisms allow you to obtain simple and complex results by mutating/transforming/stretching/compressing the metagraph quickly and cleanly.
Lastly, it is also useful to associate metagraph edges $E_i$ with nite lists $V_i$ of Values, each of which may be integer, floating-point or more complex in structure. The reason for these Values will be mentioned later.

#### 2.2.1.2 Atoms

The vertices (nodes) and edges (links) of a graph (metagraph), known as Atoms, are used to represent not only "data", but also "procedures" and then, many metagraphs are executable

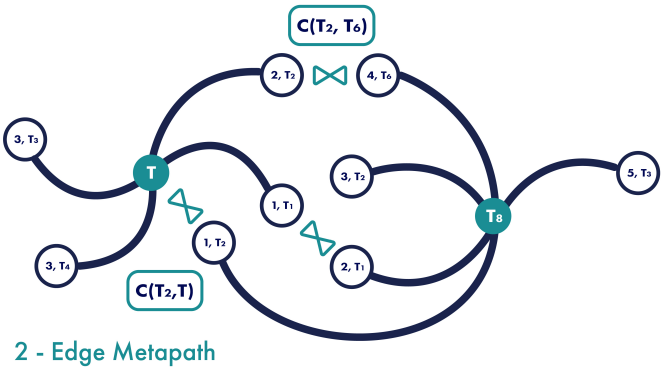



**Figure 2.7:** A short metapath, formed by connecting two directed edges.

programs as well as data structures.
Atoms are one of the main components of the AtomSpace. Atoms, together with Values are what the AtomSpace stores.
The two primary types of Atoms are Nodes and Links. They are used to represent anything that resembles a graph-theoretical graph.
From the TMG definition above, Atoms are typed (in the sense of Type Theory) and thus, they can be used to store a large variety of information. Values are used to assign "valuations" to Atoms, to indicate the truth or likelihood of that Atom, or to hold other kinds of transient data. Every Atom has a key-value database attached to it, that can store any kind of information about that Atom. The distinction between the "shape of the graph", and its related "data" is central for allowing high-speed graph traversal and generalized graph query.
Note that, the name "Atom" was chosen because of the resemblance of OpenCog Atoms to the concept of atoms in mathematical logic. Moreover, in Ruby and Prolog programming languages, symbols are literally called atoms; in both LISP and Guile, allow parameters to be attached to symbols, as Values can be attached to Atoms.

When an atom are placed in the AtomSpace, it becomes unique: it gets a single, unique ID. Thus, in comparison to other programming languages[2], Atoms can be understood to be the same thing as symbols. The AtomSpace is essentially a symbol table, which commonly has the unique-symbols property.
The atom ID is the string name of a Node and the outgoing set of a Link. Specifically: Nodes are identified by their names or labels, that is the only property that they have. Links are iden-

[2] Section 2.2.2 explains why it is possible to talk about programming languages

tified by their contents, which are ordered or unordered sets of other atoms. Links do not have any name or label other than their contents: a link is uniquely identified by it's type and it's contents.
A TruthValue (a certain type of Values) gives each Atom a valuation or an interpretation and conseguently, all Atoms in a given, fixed AtomSpace always carry a default valuation/interpretation (i.e. a SimpleTruthValue) along with them. Naturally, an Atom may have one or more different kinds of TruthValues.
All TruthValues (tv) expose at least two parameters, representative of the SimpleTruthValue (stv):

1. Strength: it represents a probability estimate of the true unknown probability. It is a floating-point value ranging from 0 to 1, with 0 denoting the classical Boolean false, and 1.0 denoting true.

2. Confidence: it captures the spread of the second order distribution over the true unknown probability. It is a floating point value ranging from 0 to 1, expressing the certainty of the strength, with 0 denoting completely uncertain, and 1 denoting completely confident.

The types form a type hierarchy: all atoms inherit from the type "Atom", and the type Atom itself inherits from ProtoAtom. The ProtoAtom is itself the base type for Values as well as Atoms. The atom type category site[3] lists all documented theorized, proposed, currently in use, deprecated and obsolete atoms.
It is not a complete list: new types are easily invented, and the various OpenCog books mention Atom types that are not implemented or have been implemented in a different way. This reflects a more general point: the specific collection of Atom types in an OpenCog system is bound to change as the system is developed and experimented with. In the current system, it does not necessarily have a profound and lasting significance.

Following a descriptive list of the main types of atoms used for this project:

- **ConceptNode**: a Node representing any concept.
  Its TruthValue, composed of at least a strength and a confidence value, has the strength that indicates the occurence of a concept within the context of experience and the confidence that indicates how sure the agent or system is of this value.

[3] https://wiki.opencog.org/w/Category:Atom_Types

For example, imagine an empty AtomSpace and a newborn agent that begins observing the world for the first time. If the first two things it sees are a man and a cat, it may define the following concepts:

**Code in Scheme Programming Language notation:**
ConceptNode example.

```
1        ConceptNode "man" (stv 0.5 0.001)
2        ConceptNode "cat" (stv 0.5 0.001)
```

Since the agent has only observed two concepts in its universe, it will split in two the universe (and Strength accordingly): half (0.5) consists of men and the other half (0.5) consists of cats. Because the agent has not made a lot of observations yet, it may assign a low Confidence to these values.

- **InheritanceLink**: a Link specify is-a relationships.
  In the OpenCog system, basic InheritanceLinks are used to specify both intensional (is-a) and extensional (is-an-instance-of) relationships.

  **Code in Scheme notation:**
  InheritanceLink example, which specifies that a cat is an animal.

```
1        InheritanceLink
2                ConceptNode "cat"
3                ConceptNode "animal"
```

  In this case, the TruthValue associated to InheritanceLink should be 0, if it is considered as extensional inheritance (inheritance between sets based on their members) and a value greater than 0 in the case of intensional inheritance (inheritance between entity-types based on their properties); obviously the value depends on how the agent assigns it.

- **PredicateNode**: it names the predicate of a relation.
  Predicates are functions that have arguments, and produce a truth value as output. Predicates in OpenCog roughly resemble the predicate of first-order logic, but more general. It is very similar to a characteristic function in probability theory, which helps assign a floating-point truth value to a declaration. Its usege will be understood at later points.
  Two interesting type of Atom that are derived from PredicateNode are **DefinedPredicateNode** and **GroundedPredicateNode**.
  The first is a single atom that is attached to a more complex definition. During the evaluation of the predicate, its definition (typically some formula or other complex expression, constituted entirely of Atoms) is looked up and is used during evaluation.
  The second specifies a predicate whose truth value is updated by the evaluation of a

scheme, python or C++ code snippet. It is a "black box" from the point of view of logical inference and knowledge representation and it is used to interface to external data systems, for example, to robot motor control systems, or to sensory input systems. It is impossible to reason with it, as opposed to DefinedPredicateNodes, which are "clear boxes" whose inner workings are visible to inference, learning and analysis algorithms.

- **ListLink**: a Link used for grouping Atoms for some purpose, typically to specify a set of arguments to some function or relation.
  The ListLink is best understood as a Cartesian product[4], because all of its uses involve passing an ordered sequence of arguments to some function or predicate. Ordered sequences can be naturally understood as Cartesian products, which is extremely general, cutting across all branches of mathematics.

- **EvaluationLink**: it provides a way for specifying the truth value of a predicate on a set of arguments.
  The EvaluationLink is the most central and important atom type in OpenCog, as it is how OpenCog implements knowledge representation.

  **Code in Scheme notation:**
  EvaluationLink general structure, followed by a practical example.

```
EvaluationLink <tv>
        PredicateNode some_p
        ListLink
                SomeAtom val_1
                OtherAtom val_2
```

  This indicates that the predicate *some_p*, applied to arguments *val*_1 and *val*_2, has the TruthValue $tv = some\_p(val\_1, val\_2)$.
  Practical example, $3 < 42$ is true:

```
EvaluationLink <true_tv>
        PredicateNode "LessThan"
        ListLink
                NumberNode 3
                NumberNode 42
```

- **QueryLink**: it is used to specify a search pattern that can be grounded, solved or satisfied. Patterns consist of a set of clauses, with each clause containing one or more variables. When executed, the pattern matcher attempts to find groundings for the variables; that is, it attempts to find values for the variables such that the resulting

[4] https://github.com/opencog/atomspace/issues/1490#issuecomment-352961503

graph exists in the AtomSpace.
QueryLink performs graph rewriting: after finding a match for a pattern, it uses the results to create a new pattern.
The QueryLink is a 2-ary or a 3-ary link, containing an optional variable declaration, followed by a pattern, followed by an implicand (consequent). In this project is used in a 3-ary form and thus, only the explicitly-declared variables are bounded.
QueryLink is treated as a graph re-write rule by the pattern matcher. That is, if the graph P (the antecedent) is found, then the graph Q (the consequent or implicand) is instantiated in the AtomSpace. If the graph Q is an **ExecutionOutputLink** atom type, such as in this project, then the specified Scheme/Python/C++/etc. routine is executed, with the rest of the ExecutionOutputLink contents passed as arguments. Conceptually, the pattern matcher acts to apply rules, specified as QueryLinks, to the contents of the AtomSpace. It implements a single step of a forward chainer.

**Code in Scheme notation:**
EvaluationLink general structure, followed by a pratical example.

```
        QueryLink
                $variable_declarations   # optional
                P
                ExecutionOutputLink      # graph Q
                        GroundedSchemaNode "some-routine"
                        ListLink
                                A_1
                                A_2
                                ...
```

The expression or predicate P is the pattern to be matched. The variables appearing in P are listed in the $variable_declarations. If a match to P is found, the variables are grounded with their matching atoms. The various A_1, A_2, etc. are created for each match, and are then passed as the first, second, etc. arguments to a function "some-routine". This function can then perform any action, for example drives external systems, such as motor controls or other devices.

There are other atom types used here. A more general list of Opencog atom types, which can be divided into categories, is the following:

- Arithmetic Atom Types: EqualLink, GreaterThanLink, NumberNode, IntervalLink, etc.
- Boolean Atom Types: AndLink, OrLink, NotLink, etc.
- All atom types used for Natural Language Processing (NLP)

- Atom types for the OpenCog Probabilist Logic Network
- Atom types to perform Pattern Matching (PresentLink, AbsentLink, VariableList / VariableSet / VariableNode, etc.)
- Atom types for the threading
- Set-theoretical Atom Types: types having some form of set-theoretical signficance (MemberLink, SetLink, SubsetLink, etc.)
- Atom types to perform searches over the AtomSpace (JoinLink, QueryLink, MeetLink, etc.)

There is a type system for working with atom types: that is, the type of an atom can itself be specified using other atoms, which also applies to the polymorphic types. The type system allows for basic error checking, such as with the type checker, and more generally, it allows type-logical reasoning and inference to be performed on atom signatures.

In conclusion, the AtomSpace Explorer web tool can be used to display OpenCog data on screen. Atomese is fetched from the AtomSpace, then displayed as a two dimensional graph in the browser, Figure 2.8 provides an example.

As closing note of this section, to greatly simplify the work in this project, uncertainty-free inference is used, so all TruthValues associated with atoms are always TRUE, that is what the robot "sees, does and thinks" has no probability of "failure". But, as amply demonstrated above, the system is set up to work in completely uncertain environments and with uncertain reasoning.
Indeed, one of the most interesting modules of OpenCog is Probabilistic Logic Networks (PLN), a conceptual, mathematical and computational approach to uncertain inference. Although this project does not use it, it is particularly suitable for uncertain reasoning, especially when knowledge is based on limited observations of reality, but it can also handle abstract mathematical reasoning, and the relationship between the two, therefore this project could be integrated into it in the future.
PLN is currently used as a basic module for reasoning algorithms [31, 32].

### 2.2.2 Atomese

Because of these many and varied Atom types, constructing graphs to represent knowledge looks like a kind of "programming".

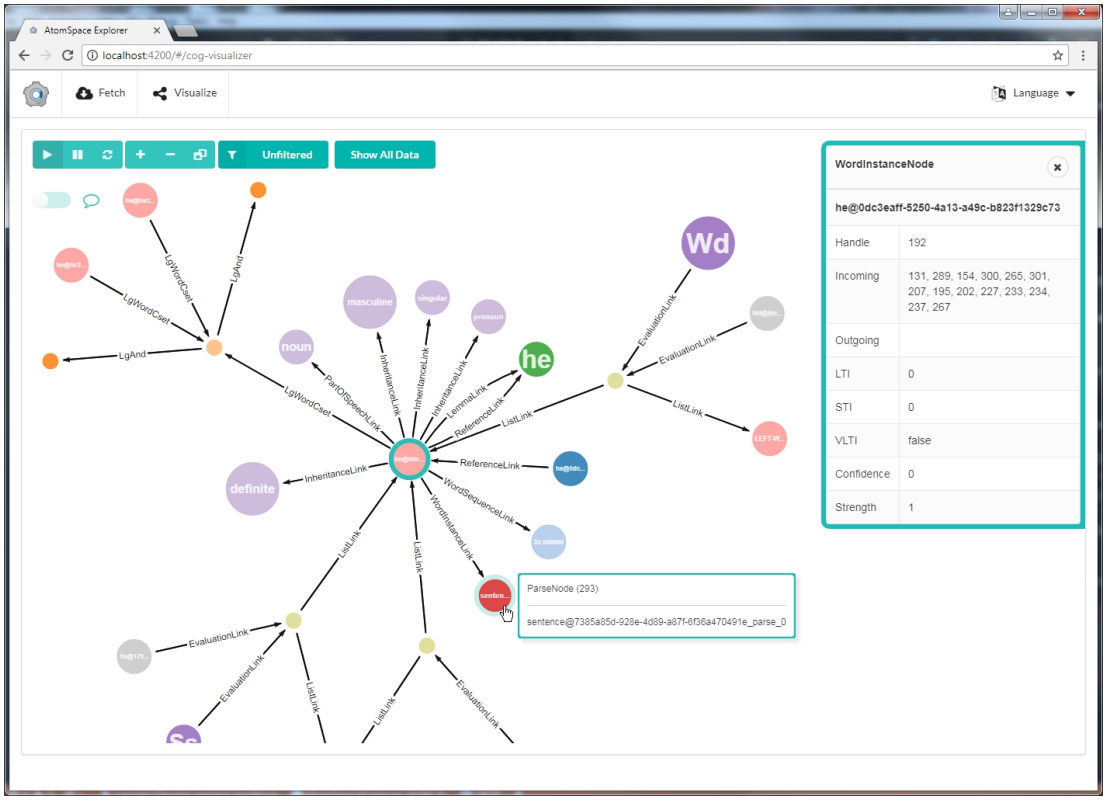



**Figure 2.8:** Atomspace example. By selecting an atom, you can get its unique ID, the type of atom (associated with an integer), incoming-set and outgoing-set, Strength and Confidence and much more.

The programming language is informally named "Atomese".
Atomese is the concept of writing programs with Atoms. It vaguely resembles a strange mash-up of SQL, due to queriability, Prolog/Datalog, due to the logic and reasoning components, Lisp/Scheme, due to lambda expressions, Haskell/CaML, due to the type system, and rule engines, due to the graph rewriting and forward/backward chaining inference systems.

Atomese is not, and was never intended to be, a programming language with which humans would write source code, like Java, Python, LISP or Haskell. Rather, it is a language that computer algorithms, such as genetic programming systems, term rewriting systems and rule engines, would be able to manipulate. It is a graph database that pattern mining algorithms could search, manipulate and transform and is designed for easy self-introspection.

In its current form, Atomese was primarily designed to allow the generalized manipulation of large networks of probabilistic data by means of rules and inferences and reasoning

systems. It extends the idea of probabilistic logic networks to a generalized system for algorithmically manipulating and managing data. The current, actual Atomese design has been heavily influenced by practical experience with natural-language processing, question answering, inferencing and the specific needs of robot control.

The use of the AtomSpace, and the operation and utility of Atomese, remains a topic of ongoing research (a re-evaluation of the OpenCog architecture is currently underway. A new OpenCog system called "Hyperon" is being studied [33], introducing upgrades from Atomese to Atomese 2.0 [34], Distributed AtomSpace architecture [35, 36, 37, 38] and much more) and design experimentation, as various AI and knowledge-processing subsystems are developed. These include machine learning, natural language processing, motion control and animation, deep-learning networks and vision processing, constraint solving and planning, pattern mining and data mining, question answering and common-sense systems, and emotional and behavioral psychological systems. Each of these impose sharply conflicting requirements on the AtomSpace architecture. The AtomSpace and Atomese are the current best-effort KR system for satisfying all these various needs in an integrated way.

### 2.2.3 Pattern Engine, Unified Rule Engine and Relex2Logic

Three more components of the OpenCog architecture are now introduced: Pattern Matcher, Unified Rule Engine and Relex2Logic.
Each of them is necessary for the following ones, in order of listing. Unified Rule Engine is mostly built on top of the Pattern Matcher and it is applicable to rules written in a Scheme/Atomese representation, such as for Relex2Logic and PLN.

#### 2.2.3.1 Pattern Matcher

OpenCog has a Pattern Matcher (PM), or Query Engine or Variable Unifier, that can be used to search the AtomSpace for specific patterns or arrangements or 'templates' of atoms. The PM can be used from C++, Scheme or Python.
After specifying some (arbitrarily complex) arrangement of atoms, that is, a hypergraph consisting of Nodes and Links of several types, the PM can find all instances of that hypergraph in the AtomSpace. The pattern or template can have "holes" in it, locations that are variable, and so the pattern matcher can act to "fill in the blanks" when presented with a pattern that has blanks in it (better known by the expression: 'grounding' a pattern).

For example, the VariableNode type are used to indicate these “blank spots”.
For a good introduction to these concepts, see [39]. Although, the PM implementation is considerably more sophisticated: it unifies multiple terms at once, automatically handles unordered terms (terms that can be in any order) and provides support for quotation, execution and evaluation.

For this project, patterns are specified using QueryLink type.
One creates the pattern-template that one wishes to search for. The pattern can be specified as a collection of trees, containing the VariableNodes, making up the graph. During the search, all matching graphs are found and the groundings are noted.
The QueryLink is used for graph-rewriting, thus these grounding are then pasted into a second pattern, to create a new graph.
In Section 3.2, practical applications will be presented to help understand these concepts as well.

#### 2.2.3.2 Unified Rule Engine

The Unified Rule Engine (URE) is a generic OpenCog rule engine operating on the AtomSpace and it can be used to implement any logic.
Two chaining modes are currently supported, Forward Chaining and Backward Chaining.
The strengths of the URE are:

- Reads/writes knowledge directly from/to the AtomSpace
- It is generic, can be used to implement any logic, even higher order logics with some limitations
- Comes with a powerful control mechanism to speed up reasoning

It was used as a first approach to solve the problem faced in this project. It was later abandoned due to conceptual and practical problems in favour of a new approach (Section 4.5), although, in the end, good ideas emerged that could solve these problems. See related Section [...sviluppi futuri].
For these reasons, any other information related to URE is referred to [40] and the webpages linked to that.

#### 2.2.3.3 Relex2Logic

This project handles a Natural Language Processing (NLP) module. Here, some components of OpenCog this module superficially uses, are very briefly presented.
The component on the top is called Relex2Logic (R2L).
R2L produces a certain form of predicate-argument structure for an English-language sentence. That structure roughly resembles classical propositional or predicate logic, thus the name.
It is pointless to go into the details of this module because, first of all, it became obsolete because its java rule engine and representation scheme were inadequate and knowledge extraction was too far removed from the reasoning and anaphora components; secondly, because the predicate-argument structure is produced by applying a set of rules to the RelEx format (see below), using the forward chainer provided by URE, which require a very good knowledge of the topic in order to be understood.

Thus, this module takes as input an English sentence and returns a relatively simple logical representation of it in Atomese. That is, a hypergraph composed of atoms that are associated with the words of the sentence and then structured to express the logic of it in the form of a hypergraph. The following example is used to better explain:

```
1         The English sentence: "The cat is an animal"
2
3 ---------------- Result:
4 (EvaluationLink (stv 1 1)
5   (PredicateNode
6     "is.v@5fdee4a0-7ad4-4aa7-9172-55a137c3dbdf"
7       (stv 9.75697e-13 0.00124844))
8   (ListLink
9     (ConceptNode
10      "cat.n@e425545c-ceea-4eee-931b-a6c312d66d58")
11    (ConceptNode
12      "animal.n@1813a95c-0c44-46f6-bad3-7232f6ff8616")))
13
14 (InheritanceLink (stv 1 1)
15   (ConceptNode
```

```
16      "cat.n@e425545c-ceea-4eee-931b-a6c312d66d58")
17    (ConceptNode
18      "animal.n@1813a95c-0c44-46f6-bad3-7232f6ff8616"))
19
20  (EvaluationLink (stv 1 1)
21    (PredicateNode
22      "is.v@5fdee4a0-7ad4-4aa7-9172-55a137c3dbdf"
23        (stv 9.75697e-13 0.00124844))
24    (ListLink
25      (ConceptNode
26        "cat.n@e425545c-ceea-4eee-931b-a6c312d66d58")))
27
28  (InheritanceLink (stv 1 1)
29    (InterpretationNode
30      "sentence@c9bf0479-2636-4a8b-901c-38d1305ed29e_
31  parse_0_interpretation_$X" (stv 9.75697e-13 0.00124844))
32    (DefinedLinguisticConceptNode
33      "DeclarativeSpeechAct" (stv 9.75697e-13 0.00124844)))
34
35  (ImplicationLink (stv 1 1)
36    (PredicateNode
37      "is.v@5fdee4a0-7ad4-4aa7-9172-55a137c3dbdf"
38        (stv 9.75697e-13 0.00124844))
39    (PredicateNode "be" (stv 9.75697e-13 0.00124844)))
40
41  (InheritanceLink (stv 1 1)
42    (ConceptNode "animal.n@1813a95c-0c44-46f6-bad3-7232f6ff8616")
43    (ConceptNode "animal"))
44
45  (InheritanceLink (stv 1 1)
46    (ConceptNode "cat.n@e425545c-ceea-4eee-931b-a6c312d66d58")
47    (ConceptNode "cat"))
48
49  (InheritanceLink (stv 1 1)
50    (PredicateNode
51      "is.v@5fdee4a0-7ad4-4aa7-9172-55a137c3dbdf"
```

```
52          (stv 9.75697e-13 0.00124844))
53      (DefinedLinguisticConceptNode
54          "present" (stv 9.75697e-13 0.00124844)))
55
56  (EvaluationLink (stv 1 1)
57      (DefinedLinguisticPredicateNode
58          "definite" (stv 9.75697e-13 0.00124844))
59      (ListLink
60          (ConceptNode
61              "cat.n@e425545c-ceea-4eee-931b-a6c312d66d58")))
```

All the Atoms whose names contain the @ character are those that are associated with the words in the sentence. In this way, the same word in different places can take on different meanings and links.
However, these atoms (such as *cat.n@...*, *animal.n@...* and *is.v@...*) are actually inherited by the concepts (ConceptNodes) that represent them, so 'cat', 'animal' and 'is' (lines 35, 41, 45).
Moreover, the atom in line 4 perfectly describes the relationship between these three concepts in the sentence.

The processing pipeline for a sentence is:

1. Sentence ⇒ Link-Grammar
2. Link-Grammar ⇒ RelEx
3. RelEx ⇒ Relex2Logic

Step 1 performs a syntactic parse of a sentence, using the Link Grammar (LG) parser.

Step 2 converts the LG output into a dependency grammar (DG) format that is more closely aligned with de-facto DG standards. This includes the unpacking of some of the LG output into standard linguistic feature classifications, e.g. for person, number, tense, aspect, mood, voice.
This converter is called RelEx, a Narrow-AI component of OpenCog. It is an English-language semantic dependency relationship extractor, built on the Carnegie-Mellon Link Grammar parser. It uses a series of graph rewriting rules to identify subject, object, indirect object and

many other syntactic dependency relationships between words in a sentence. That is, it generates the dependency trees of a dependency grammar.
Unlike other dependency parsers, RelEx attempts a greater degree of semantic normalization: for questions, comparatives, entities, and for modifying clauses, and sub-ordinate clauses.

Step 3 extracts the predicate-argument logical structure, as shown above.

# 3
# Problem Description

The problem addressed in this project is based on the classic blocks world problem[1]. The blocks world is a planning domain in artificial intelligence and is used as a toy problem. From an algorithm perspective, it is an NP-Hard search and planning problem. Here, a more general version of it is presented.
First of all, the environment consists of:

- A robot manipulator, equipped with a video camera
- Blocks (cubes) in the same size, which all have an AprilTag [41, 42] attached to one of the faces
- Zero or more fixed objects, also with AprilTags

Figure 3.1 illustrates a possible configuration of the environment, which is the one used in this project.
Compared to the state-of-the-art, where the robot arm has to pick and place the blocks and the goal is to build one or more vertical stacks of them, the goal is generalized in a way that any arrangement of the blocks can be achieved. That is, given any initial arrangement of the blocks in the environment, find the shortest sequence of actions that the robot has to perform in order to achieve the required arrangement.

[1] https://en.wikipedia.org/wiki/Blocks_world#:~:text=In%20its%20basic%20form%2C%20the,different%20sizes%2C%20shapes%20and%20colors.

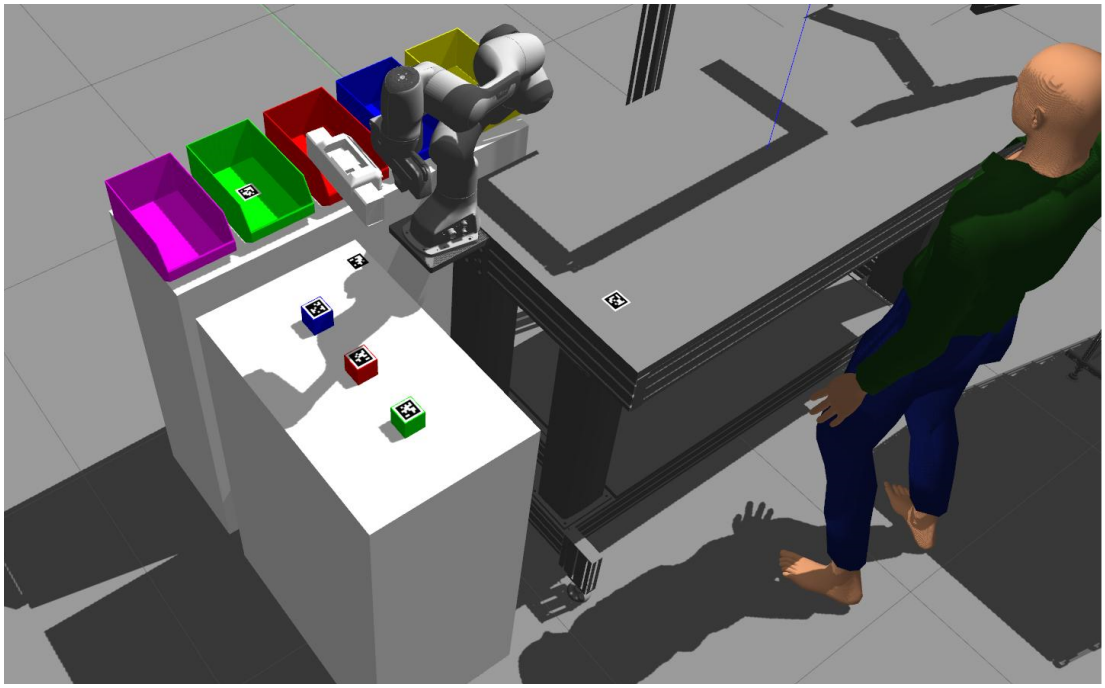


**Figure 3.1:** The configuration of the environment used in this project: a white table on the left, a grey workbench on the right, on which the robot manipulator is fixed, and behind it, five bins of various colours, which will contain the different blocks. All these objects have an AprilTag on top of each one.

In order to define the implementation, it is first necessary to explain the constraints of this problem:

- The robot knows only four actions: *Pickup*, *Putdown*, *Stack*, *Unstack*.
- Each block is *clear* if and only if it has no object on it and is not *in-hand* (i.e., the robot's hand can take it with a single action).
- Each block can be *on* the table, on top of another object or *in-hand*
- The robot's hand may be busy (it is holding a block) or free (it holds nothing)
- Only one block can be picked up at a time, putting it down before the next. It is not possible to pick up a block that is under another one (because it is not considered *clear*) or to move a fixed object.

The *Pickup* action is the opposite of *Putdown* and they are used to take/place a block from/on a fixed object. The *Stack* action is the opposite of *Unstack* and they are used to put/take a block on/from another block or a fixed object.

Basically these actions mirror physics.
For example, consider two blocks (A and B) on a table and block A is *on* block B: to pick up block B, that block must be *clear* and the hand must be free. Initially, block A is *clear*, while B is not. Thus, one can do *Unstack* of blocks A and B, getting block B *clear* and block A *in-hand*, then do *Putdown* of block A and finally do *Pickup* of block B, achieving the goal. Figure 3.2 illustrates these steps.

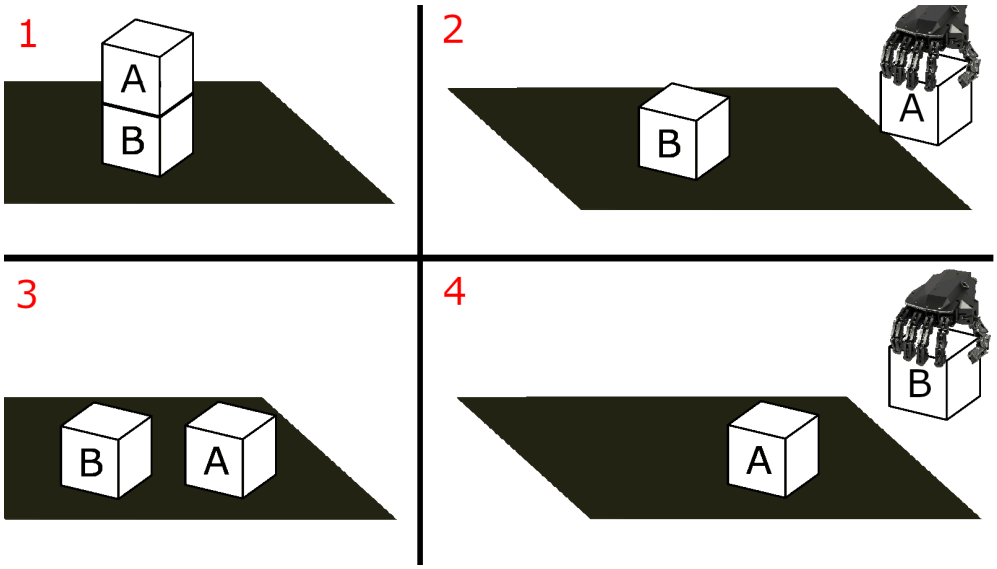



**Figure 3.2:** Three actions to go from state 1 (initial arrangement) to state 4 (target arrangement): *Unstack* of A and B, *Putdown* of A, *Pickup* of B.

Since it is possible to have more fixed objects (such as tables, workbench, etc.) than in the blocks world, where there is only a work plane, any free block can always be considered on top of something else. Consequently, the actions *Pickup* and *Putdown* lose their meaning because they become subcases of the respective *Unstack* and *Stack*. However, in this project all four are still maintained and used, as explained below.

## 3.1 Knowledge Base in Atomese

Any aspect of the environment, the problem, the robot or anything else can be represented in the form of atoms, which will populate the AtomSpace (i.e. the hypergraph of the KB). There are many different ways of encoding these aspects. The one used in this project is pro-

posed below.

Firstly, the Atomese representation of the objects:

```
(InheritanceLink
        (ConceptNode "screw")
        (ConceptNode "object"))

(InheritanceLink
        (ConceptNode "table")
        (ConceptNode "fixed-object"))
```

On a practical (physical) level, the objects will all be cubes, except for the fixed ones. At the conceptual level, a semantic meaning will be associated to the AprilTags of each cube. In this way, and thanks to the AprilTags, it is possible to simplify the perception and manipulation of objects by replacing each different object with a cube.
In other words, it will be possible to "simulate" working with any kind of object, and thus demonstrate the potential of the NLP aspect of this project, leaving out perception and manipulation, which do not concern it at the moment.
For this reason, the code above speaks of *screw* and not *cube_1*.

In that code, the *screw* has been defined as an *object* and the *table* as a *fixed-object*. ConceptNodes were used to refer to "concepts" and InheritanceLinks to connect two ConceptNodes with an "inheritance" meaning. This way, when the robot searches for which objects it can move, it will limit the pattern matching only to atoms that are objects (thus, only to the screw).
As each atom is unique, it is not necessary to first define objects (i.e., *screw* and *table*) and concepts (i.e., *object* and *fixed-object*) and then, link them. Within the same AtomSpace, once an atom is defined, whether independently or within any hypergraph, it will remain unique and the creation of other hypergraphs containing that atom will not create a new one, but rather it will be shared between the various hypergraphs, in a sense, connecting them.

In addition, by construction of pattern matching rules, it is necessary to explicitly define the "non-equality" between all the various pairs of objects. Therefore, for example, given three objects such as *screw*, *bolt* and *table*, there will be atoms like these:

```
(NotLink (EqualLink
        (ConceptNode "screw") (ConceptNode "bolt")))

(NotLink (EqualLink
        (ConceptNode "screw") (ConceptNode "table")))

(NotLink (EqualLink
        (ConceptNode "bolt") (ConceptNode "table")))
```

They correspond to all possible simple combinations (without repetition) that can be constructed from the set of all objects (fixed ones included) taken two at a time.

Finally, now that the objects have been constructed, inherited and differentiated, one or more states must be associated with them. The best way to represent the state of an object is via an EvaluationLink, which associates a PredicateNode with one or more objects (their relative atoms). The possible states of an object are: *clear*, *in-hand*, *on*. Thus, for example:

```
(EvaluationLink
        (PredicateNode "clear")
        (ConceptNode "bolt"))

(EvaluationLink
        (PredicateNode "in-hand")
        (ConceptNode "screw"))

(EvaluationLink
        (PredicateNode "on")
        (ListLink
                (ConceptNode "bolt")
                (ConceptNode "table")))
```

In Appendix A, Section 7.1, a complete examples have been added for further details.

## 3.2 Actions in Atomese

Once the atoms relative to the objects have been defined, the robot's actions must also be defined as hypergraphs. From this point on in the paper, these actions will also be called "rules", being called rules the ones that are executed within AtomSpace for pattern matching.
Before exposing them, Section 7.2 in Appendix A, provides the domain and problem in Planning Domain Definition Language (PDDL) notation of the standard block world problem. Automated planning and scheduling problems are usually described in the PDDL notation, which is an AI planning language for symbolic manipulation tasks. Starting from those definitions of the various actions, one can better understand how they are created in Atomese.

Each action used in this project, as in the PDDL notation, requires preconditions and applies effects. Below is a list of these, for each of them:

- **Pickup** a block from the intial fixed object.
  Preconditions:
  - block must be clear
  - block must be an object
  - block must not be "stacked"

  Effects:
  - block no longer clear
  - block is in hand

- **Putdown** a block from the hand to a fixed object.
  Preconditions:
  - block must be in hand
  - block must be an object

  Effects:
  - block is clear
  - block no longer in hand

- **Stack** block A on top of the block B.
  Preconditions:
  - block A must be different from block B
  - block A must be an object
  - block A must be in hand
  - block B must be a clear object or must be a fixed object

  Effects:
  - block A is on top of block B
  - block A is "stacked"
  - block A no longer in hand
  - block A is clear
  - block B no longer clear (if it is an object)

- **Unstack** block A from the top of block B.
  Preconditions:
  - block A must be different from block B
  - block A must be an object
  - block A must be clear
  - block A must be on top of block B
  - block B must be an object or a fixed object

  Effects:
  - block A is in hand
  - block A no longer clear
  - block A no longer on top of block B
  - block A no longer "stacked"
  - block B is clear

Only the *Pickup* rule is presented and explained here, as an example, whereas the code of the other three can be found in Appendix A, Section 7.3.

**Code in Atomese notation:**

*Pickup* action rule.

```
(define (pickup-action . args)
  (define precond (car (cdr args)))
  (define eff (car args))
  (cog-extract-recursive! precond)
  eff
)

(define (pickup args)
  (let
    ((rule
      (if (equal? args '())
        (QueryLink
          (VariableList
            (TypedVariableLink
              (VariableNode "?ob")
              (TypeNode "ConceptNode"))
          ) ; parameters
          (AndLink
            (PresentLink
              (InheritanceLink
                (VariableNode "?ob")
                (ConceptNode "object"))
              (EvaluationLink
                (PredicateNode "clear")
                (VariableNode "?ob"))
            )
            (AbsentLink
              (EvaluationLink
                (PredicateNode "stacked")
                (VariableNode "?ob"))
            )
```

```
32            )
33            (ExecutionOutputLink
34              (GroundedSchemaNode "scm: pickup-action")
35              (ListLink
36                ; effect
37                (EvaluationLink
38                  (PredicateNode "in_hand")
39                  (VariableNode "?ob"))
40                ; precondition
41                (EvaluationLink
42                  (PredicateNode "clear")
43                  (VariableNode "?ob"))
44              )
45            )
46          )
47          (QueryLink
48            (VariableList
49              (TypedVariableLink
50                (VariableNode "?ob")
51                (TypeNode "ConceptNode"))
52            ) ; parameters
53            (AndLink
54              (PresentLink
55                (InheritanceLink
56                  (VariableNode "?ob")
57                  (ConceptNode "object"))
58                (EvaluationLink
59                  (PredicateNode "clear")
60                  (VariableNode "?ob"))
61              )
62              (EqualLink
63                (VariableNode "?ob")
64                args
65              )
66              (AbsentLink
67                (EvaluationLink
```

```
68                  (PredicateNode "stacked")
69                  (VariableNode "?ob"))
70              )
71            )
72            (ExecutionOutputLink
73              (GroundedSchemaNode "scm: pickup-action")
74              (ListLink
75                ; effect
76                (EvaluationLink
77                  (PredicateNode "in_hand")
78                  (VariableNode "?ob"))
79                ; precondition
80                (EvaluationLink
81                  (PredicateNode "clear")
82                  (VariableNode "?ob"))
83              )
84            )
85          )
86        )
87      ))
88      rule
89    )
90  )
```

The code is divided into two functions: *(pickup-action . args)* and *(pickup args)*.
The main rule (or action or function) is the second one, called *pickup* (line 8), which describes the action of picking up some object. It is composed of two large QueryLink atoms (lines 12 and 47), as for all other actions.
Only one of the two atoms is used for each call to this function, through the 'if' conditional construct (line 11). It checks whether the function has an argument *args* other than empty *'()*. If it is indeed empty, then the QueryLink from line 12 to 46 is used, otherwise the one from line 47 to 85, which actually uses the *args* argument. Indeed, the variable *rule* is defined as one of the two QueryLinks, depending on the result of the conditional construct, and then, it is applied in line 88.
These QueryLinks are completely identical, except for the EqualLink atom on line 62. Consequently, take a look first at the QueryLink without arguments, based on the generic de-

scription of a QueryLink in Section 2.2.1.2.

Its variable_declaration part consists of a *?ob* parameter (VariableNode) that is restricted to the ConceptNode type (lines 13-17). In this way, the rule will do pattern matching using the conditional part as a pattern, which will be parameterized by *?ob* that will result as all possible objects that can be picked up, given the current composition of the AtomSpace. Thus, the notation "object (parameterized)" is given to generic objects that satisfy the parameterization requirements.

The conditional part (or pattern) requires that the logical AND (AndLink) be evaluated to TRUE (lines 18-32). This can only happen when both PresentLink and AbsentLink are evaluated to TRUE at the same time.

- AbsentLink is a trick to differentiate atoms simply *clear* from those *clear*, but also *on* some other object; this corresponds to differentiating *Pickup* and *Unstack* actions (design choice).
  Briefly, when an object is subjected to a *Stack* action, a "stacked" state atom is created, and when it is subjected to *Unstack*, that atom is removed from AtomSpace.
  Finally, AbsentLink checks for the absence of that atom in AtomSpace: if the object (parameterized) is simply *clear*, then the atom is not there and it can be picked up, otherwise the object (parameterized) is *on* something else and the *Unstack* action must be used instead of *Pickup*.
- PresentLink, the opposite of AbsentLink, checks for the presence of atoms composing it in the AtomSpace , just like logical AND, returns TRUE only when all are found.
  In this case, it verifies that the object (parameterized) is indeed an *object* and that it is also *clear*, from the definitions of the Pickup action.

The last part of the QueryLink (lines 33-45) is relative to the graph re-write and uses an ExecutionOutputLink atom to execute a GroundedSchemaNode. The GroundedSchemaNode type atom behaves like the GroundedPredicateNode described in the previous section, but it is not associated with an EvaluationLink, it simply executes a Schema or Python code or Combo procedure. The preference of GroundedSchemaNode, over GroundedPredicateNode, is a design choice to follow the PDDL definition of the various actions.
In this project, it was decided to use a Schema code (line 34, 'scm' stands for Schema) that executes the *pickup-action* function, with a list of arguments provided by the ExecutionOutputLink.

This list of arguments (line 35) is made up of the effect and precondition of the Pickup rule. Thus, the rule wants the object (parameterized) to be *clear*, as a precondition, and the effect corresponds to the atom describing the *in-hand* state of the object . Initially, a QueryLink structure was sought that would also work using inference, specifically backward chaining. Thus, only one atom each was accepted for preconditions and effects. Although ExecutionOutputLink uses a ListLink, which can contain multiple arguments, this design was retained considering a possible integration of an inference algorithm to solve the problem. Consequently, the remaining effects and preconditions not included in the ListLink are added and removed in the called function.
The *pickup-action* called function (lines 1-6) takes these two atoms as arguments, removes the one corresponding to the precondition from the AtomSpace and finally applies the effect, creating the corresponding atom in the AtomSpace (some other functions add more effects and remove more preconditions). This is how the action takes place inside the AtomSpace: states that would no longer be "valid", after the action has been performed ,are removed (such as the state of *clear*, because the object (parameterized) will be in hand, thus no longer *clear*) and those created due to the action are added (such as the state of *in-hand*).

Now that the first QueryLink has been explained, the second behaves in the same way, but with an additional constraint in the conditional part.
This constraint is the EqualLink in line 62 and it is used to limit the parameter (the VariabileNode) to coincide with the *args* argument, which will be an atom corresponding to the object to be picked up, effectively restricting the pattern matching to that single atom.
In practice, the QueryLink without argument performs the Pickup action on all objects that can actually be picked up, instead of the QueryLink with argument that does *Pickup* only of the object passed as argument. That is, the first shows which objects are pickable, the second actually picks one of them.
The reason why these two possibilities (QueryLinks) are necessary will be better understood when the algorithm is discussed (Section 4.5.1.2).

# 4

# The Algorithm

Having defined the OpenCog system and part of the implementation of the problem in the previous chapters, it is now possible to combine all the various modules and explain the functioning of the internal design and the main algorithm.

It is possible to divide the entire project into 6 phases: initial, perception, learning, request, search and results execution.

## 4.1 Initial Phase

The initial phase concerns the initialization of the simulated environment. The framework used for the development and programming of the robot is the well-known ROS, Melodic version, and the Gazebo 9 system handles the simulation.
Therefore, the world as in Figure 4.1 is loaded into Gazebo. It consists of a robot manipulator with an attached camera, some fixed objects (such as tables, workbench and bins) and cubes, both with their AprilTags used for detection and semantic assignment.
Then, a server in C++ code is activated and the nodes of the ROS system for robot control and object perception become available for listening.
Finally, the main AtomSpace is created and the actions rules are loaded into it.

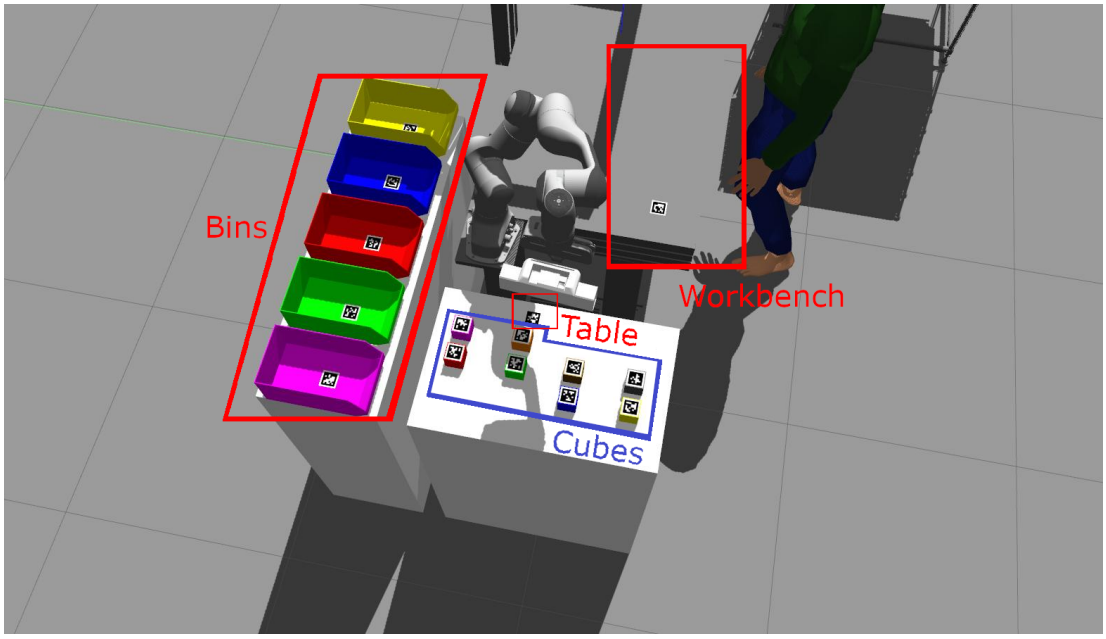



**Figure 4.1:** In the red boxes are collected the objects that are considered fixed. Meanwhile, the blue box contains the remaining objects, represented by cubes.

## 4.2 Perception Phase

The perception phase, like the remaining phases, is written in Python code and sends commands via a Python client to the C++ server mentioned above. These commands move the robot and allow it to scan its surrounding area for possible objects (related AprilTags are recognised), thanks to the camera attached to the top of it.
Of each AprilTag found, its associated semantic representation (i.e. English word describing the object, as well as the name of its relative ConceptNode, thus, for example: "table", "workbench", etc.) is returned by the server to the client, which in turn returns it to the main Python code that begins the initialization of the objects list, found in the environment.

## 4.3 Learning Phase

The learning phase is used to provide additional information.
The perception algorithm is extremely simple and does not deal with information processing. Consequently, it is not able to distinguish an object from a fixed object or to infer that an object is placed on top of another. Surely, with a perception processing module all of this, and much more, could be managed.

However, this project shows the simplicity of integration and the potential of the learning with the OpenCog system.

This phase was created to tell the robot which objects are fixed, which are on top of others (fixed objects excluded) or which one is in the robot's hand. That is, to provide the necessary additional information, which the simple perception phase does not perceive, to describe the initial situation of the environment. This is done by a module using NLP, consisting of the Relex2Logic OpenCog module and a post-processing phase.

Additional information is provided to the system via English sentences.
This module applies Relex2Logic to these sentences, creating the hypergraph describing them within a new empty AtomSpace (process explained in Section 2.2.3.3).
Next, the post-processing phase loads the rules presented in Section 7.4 and executes them within that AtomSpace. Those rules look for patterns corresponding to the possible information that the system might need and create summary atoms that can be easily used by the algorithm. Thus, from the results obtained by Relex2Logic, they infer atoms such as the ones shown in Section 3.1, which are then added to the main AtomSpace.

Moreover, from the information learned in this phase, the *clear* state atoms of the effectively free objects and the *object* inheritance atoms of the non-fixed objects are created.
Finally, the last atoms of the initial KB are added to the main AtomSpace. They are all possible simple combinations (without repetition) from the objects list taken two at a time (explained in Section 2.2.3.3, too).
The main AtomSpace is now complete and fully describes the initial situation.

## 4.4 Request Phase

The request phase behaves like the previous one, except for the aim. Whereas before, English sentences are used to give additional information, now new English sentences are requested to define the goal to be achieved.
In order to define the goal it is not necessary to describe the final complete arrangement of objects in the environment, but just the states of the objects one wants to achieve.
Thus, the same steps used in the learning phase are performed here. That is, the English sentences are processed to obtain the corresponding set of atoms that describes them. Then,

these atoms are added to a goal list used by the algorithm to find a solution.

## 4.5 Search Phase

The search phase is the core of the algorithm and tries to achieve an arrangement of objects in which all atoms in the goal list are satisfied. To find a solution, the AtomSpace of the final arrangement must contain every atom within the goal list.

Since each atom is unique in the AtomSpace, two AtomSpaces describing the same arrangement and state of objects have the same atoms, thus they are identical. Consequently, each possible arrangement of objects in the environment can be associated with one and only one AtomSpace, making it a de facto state of a Finite State Machine (FSM). Moreover, each of the four actions available to the robot can be seen as a step of this FSM. Starting from a certain state, performing an action leads that state (i.e. that AtomSpace) into a new one. However, from each state, one or more actions can be performed and thus, one or more states can be reached.

One of the best ways to handle this structure is through a tree that has AtomSpaces as tree-nodes (to distinguish them from Nodes in Atomese) and available actions as tree-edges. In addition, each tree-node has a label describing the action used, and on which objects, by its parent to create it. In this way, each branch of the tree describes a different sequence of actions. They can be derived by starting with a solution tree-node and reading the labels of the parent tree-nodes backwards until the root, adding each label to a list. Finally, reversing the list.

After the request phase, the root of the tree, containing the main AtomSpace and a *NULL* label, is created.
Next, a search algorithm, based on the well-known Breadth-First Search (BFS) (more details in [43, 44]), expands the tree in search of a solution.

### 4.5.1 BFS-Based Algorithm

The BFS-based algorithm is recursive and accepts as arguments a tree-node and a variable, which limits the actions to be performed.

Following the rules of the problem (Secion 3), it is trivial to deduce that the actions alternate for each step of the algorithm. That is, starting from the root:

1. If there exists an atom, within the AtomSpace contained in the root, that describes the *in-hand* state of an object, then only *Stack* and *Putdown* actions can create children. Conversely, if there are no objects in hand, only the actions of *Pickup* and *Unstack* can. This is because the current state of the robot's hand limits the action used to go to the next state. If the hand is free, it can only grab an object, while if it is busy, it can only place the object down.
2. From this first intuition, follows the one applicable to each step: given a certain tree-node, if the action *Stack* or *Putdown* created it, then the children of that node can only be created by *Pickup* or *Unstack* actions, otherwise vice versa.

This is the reason why the BFS-algorithm has a variable as the second argument, it holds the action used to create the node in the first argument.
Trying all the actions at each step would also give the same result, but additional atoms would have to be added to the rule pattern to check whether the hand is free or busy. Instead, following this design simplifies the rules, speeds up the algorithm and avoids unnecessary code execution.

#### 4.5.1.1 Termination Criteria of the BFS-Based Algorithm

Some additional parameters are defined to limit and improve the search. As a result, it is possible to use 4 different searches:

1. Search until a first solution is found
2. Search restricted to a maximum number of iterations
3. Explore the whole tree
4. Quick search

The quick search finds a solution, decreasing the number of iterations in most situations. As this type of problems suffers from combinatorial explosion[1], the number of tree-nodes grows exponentially as the tree expands.
One way to reduce this expansion is to limit the available actions to only those that interact

[1] https://en.wikipedia.org/wiki/Combinatorial_explosion

with one or more objects appearing in the goal list. Many unnecessary actions are avoided, however this makes the algorithm incorrect, since the certainty of finding a solution is lost. It still works well for most initial arrangements and goal lists and if it fails, the search in point 1 (which is correct if a solution exists, like searches in points 2 and 3) can be used.

Finally, a solution is found when all the atoms composing the goal list are within the AtomSpace of a tree-node.
Furthermore, the first solution found will certainly be the optimal solution, i.e. the solution that uses the minimum number of actions to go from the initial arrangement of objects to the goal one. This is because for each tree-node, before being expanded, its AtomSpace is compared with the one of any tree-node belonging to its same branch. If a match is found (i.e. they are identical), then the algorithm has come back to an arrangement of objects it has already encountered and thus, the sequence of subsequent actions would be a repetition of that associated with the matched tree-node. Consequently, the expansion of this tree-node ends.

#### 4.5.1.2 Steps of the BFS-Based Algorithm

From the definitions above, it is now possible to list the steps that the BFS-based algorithm goes through.
The first recursive call has as arguments the root node (containing the main AtomSpace) and the variable defined according to the state of the hand. Inside each call:

1. It is checked if the maximum number of iterations is reached (in the case of limited search). As a consequence, the algorithm returns or continues.

2. It is checked if the AtomSpace, contained in the tree-node argument, satisfies all the atoms in the goal list. If satisfied, this tree-node is added to the solution list and the algorithm returns or continues depending on the type of search.

3. It is checked that the AtomSpace, contained in the tree-node argument, has not been encountered before. If encountered, this tree-node is not expanded and the algorithm moves on to the next one (Go back to point 1), otherwise its expansion begins.

4. Now that the checks have been passed, the expansion of the tree-node argument occurs.
   According to the variable argument, only two of the four actions can be executed. Thus, from the AtomSpace of the current tree-node, two AtomSpace children (two

copies of the parent) are created, one for each action.
Then, for each action (rule), the respective QueryLink without argument (i.e., the one that returns all possible objects that match the pattern; i.e., the objects on which that action can be performed), is executed within the AtomSpace assigned to it.
It is necessary to use AtomSpace copies because rule execution adds and removes atoms within AtomSpace and thus, modifies it. This means that executing a QueryLink without arguments modifies the AtomSpace as if the related action had been executed on all the matched objects at the same time, leading the AtomSpace into an incorrect state.
To summarise:

(a) AtomSpace copies are created

(b) Rules on generic objects are executed, one in each AtomSpace

(c) The objects, returned as solutions of the rules, correspond to the objects on which actions can be executed. They are stored in a list for each action.

(d) AtomSpace copies are cleaned up and deleted.

5. Finally, for each object found, a copy of the AtomSpace of the tree-node argument is created again. Based on the list to which the object belongs, the corresponding action is executed in that AtomSpace, but this time using the QueryLink with the argument, which corresponds to that object. That is, the action is only executed on that object. The resulting AtomSpaces are associated with new tree-nodes and become the children of the tree-node argument.

6. At last, these new tree-nodes, obtained by executing the two actions available to each object on which they are applicable, are then added to a FIFO queue. This queue is used for BFS.
The expansion of this tree-node is finished and the tree-node at the top of the queue is extracted and passed to a new recursive call of this algorithm, together with the variable describing the action used previously in its branch.

## 4.6 Results Execution Phase

At the end of the search phase, the solution list contains the tree-nodes having an AtomSpace that satisfies all the atoms in the solution list. As already explained in Section 4.5, from each solution tree-node, a list containing a sequence of actions can be extracted. If the solution

list is not empty, then the first result corresponds to the optimal solution, otherwise either there is no solution or the search parameters need to be changed.
In conclusion, the optimal solution is sent from the client to the server, which will command the robot to perform each of those actions in sequence, thus achieving the required arrangement of objects.

## 4.7 Practical Example

The following steps provide a simple practical example in order to clarify and contextualise the above phases.

1. Initial Phase:
   Loading environment (Ros, Gazebo, Relex2Logic Server, Cubes, AprilTags, etc.)

2. Perception Phase:
   Assume that four AprilTags are detected. Assume also that the AprilTags associate tree objects with a semantic meaning corresponding to parts of an industrial product to be assembled and the fourth with "workbench". For simplicity in this example, the parts will be called "screw", "bolt" and "washer".

3. Learning Phase:
   The only additional information needed by the robot is: "The workbench is a fixed object.". From this sentence the following atom will be extracted:

```
(InheritanceLink
        (ConceptNode "workbench")
        (ConceptNode "fixed-object"))
```

4. Request Phase:
   Assign an easy task: "The washer is on the screw. The bolt is on the washer.". From these sentences the following atoms will be extracted:

```
(EvaluationLink
        (PredicateNode "on")
        (ListLink
                (ConceptNode "washer")
                (ConceptNode "screw")))

(EvaluationLink
        (PredicateNode "on")
        (ListLink
                (ConceptNode "bolt")
```

```
11                                  (ConceptNode "washer")))
```

5. Search Phase:
   The Figure 4.2 shows the first steps of tree expansion using the BFS-based algorithm. First, the hand is free, thus the algorithm will try to do *Pickup* and *Unstack* actions. Since *screw*, *bolt* and *washer* are *clear* objects, the first iteration of the algorithm will create 3 tree-nodes corresponding to the initial object arrangement (i.e., the arrangement of the root), except the one to which *Pickup* is performed.
   The *workbench* is a fixed object, thus it is not found as a solution of *Pickup*, moreover the action *Unstack* returns an empty result because there are no stacked objects[2].
   Subsequently, these tree-nodes are analysed following the order corresponding to the blue numbers in the figure. None of them is a solution tree-node or has an AtomSpace already encountered, thus, their children are created.
   Tree-node number 4 is obtained by making *Pickup* of A and then *Putdown* of the same, so its AtomSpace will correspond to the one of the root. Consequently, its expansion ends.
   The expansions continue until the yellow tree-node is reached, which has an AtomSpace containing all the atoms in the goal list. Therefore, a solution is found and the search ends (Quick search).

   The sequence of actions from the initial arrangement to the goal arrangement can easily be read from the tree-edges, going up the branch from the yellow tree-node to the root.

6. Results Execution Phase:
   The following sequence of actions will be performed by the robot, satisfying the initial request:

```
1          - Pickup washer
2          - Stack washer on screw
3          - Pickup bolt
4          - Stack bolt on washer
```

[2] The objects are actually stacked relative to the workbench. However, this fixed object can be considered as the initial one and therefore one can ignore the stacks and use actions of *Pickup* and *Putdown*, alternatively the stacks can be added as information during the learning phase and consequently, actions of *Stack* and *Unstack* will be used instead.

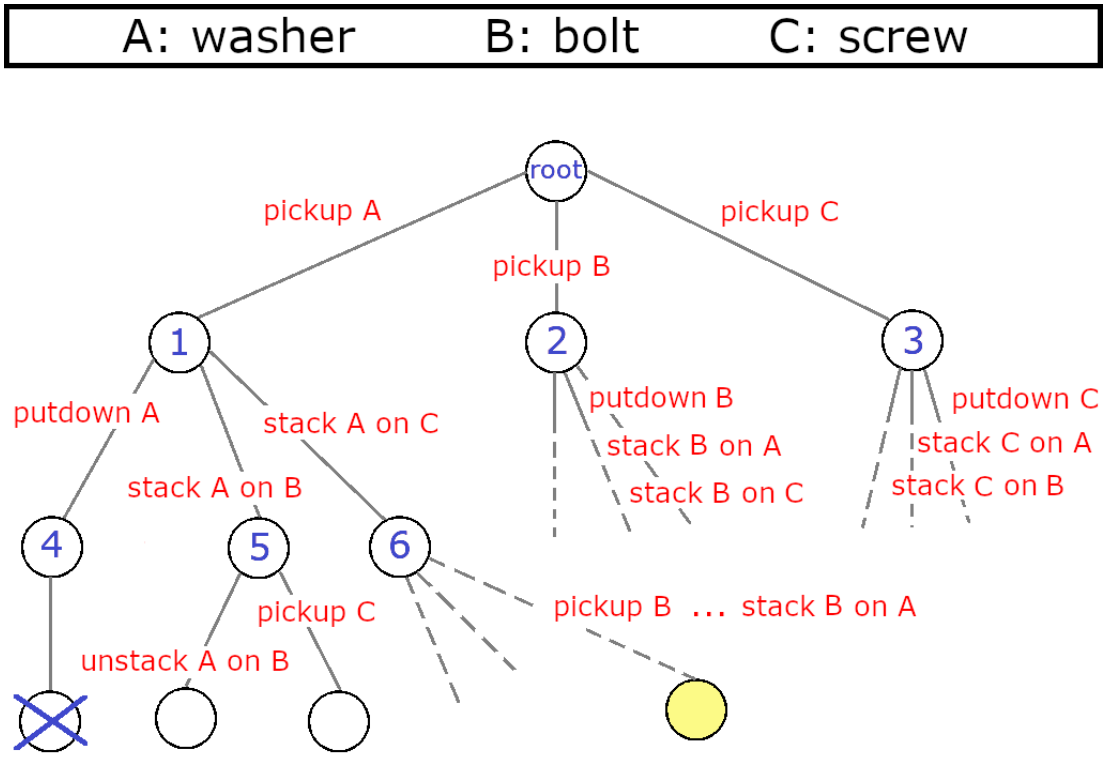


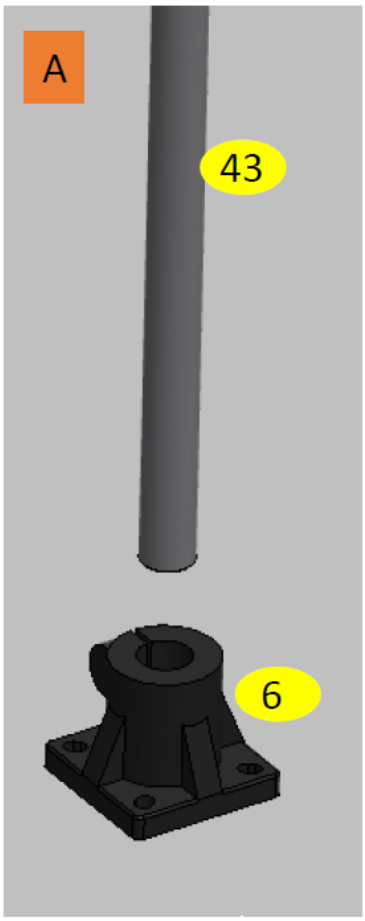

**Figure 4.2:** Graph describing the expansion of the tree of the practical example.
Each node contains an AtomSpace and each edge describes an action.
Each AtomSpace at an internal node is unique among all those contained in its branch. With the exception of AtomSpaces at leaf nodes that either already exist within its own branch (node marked with a blue X) or are a solution (yellow node).
The order of node expansion is shown by the sequence of numbers within them and follows the order of the BFS.
The legend on the top simplifies the labels in the tree by using the letters A, B, C instead of names used in the algorithm.

# 5

# Tests and Results

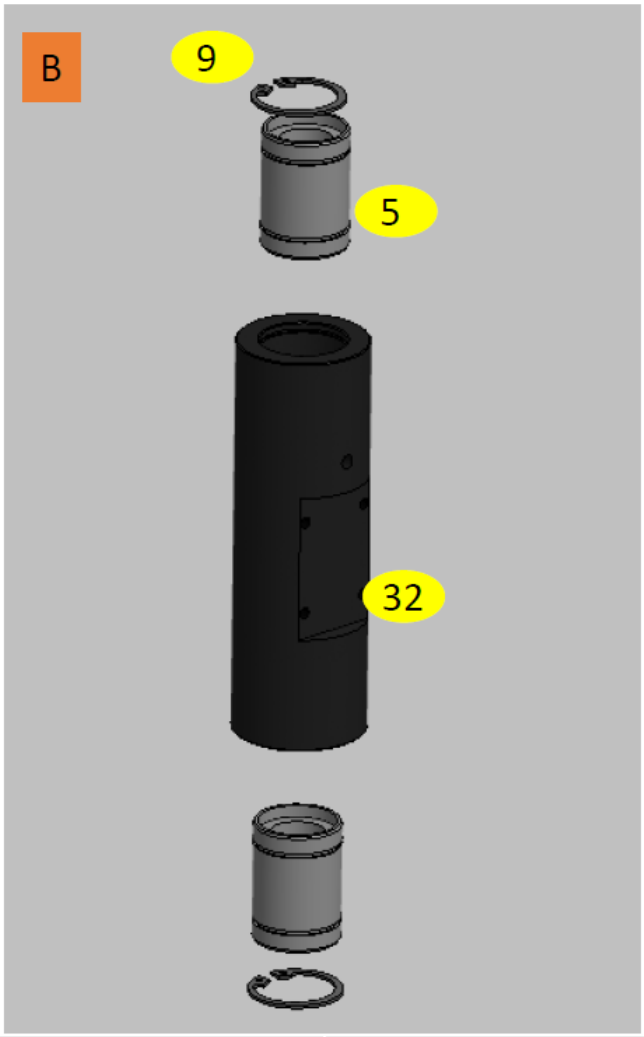
**Table 5.1:** Tabella Approcci

| | Approcci | | | | | | |
|---|---|---|---|---|---|---|---|
| **Dati di Input** | **1°** | **1°** | **1°** | **2°** | **3°** | **4°** | **5°** |
| Depth | ✓ | | | ✓ | | ✓ | ✓ |
| Conf | | ✓ | | ✓ | | ✓ | ✓ |
| Color - R | | | | | ✓ | ✓ | ✓ |
| Color - G | | | | | ✓ | ✓ | ✓ |
| Color - B | | | | | ✓ | ✓ | ✓ |
| Grey-Scale | | | ✓ | ✓ | | | |




| | | |
|---|---|---|
| 6 | 00.00.04.0127 | MORSETTO BETT Ø20mm - cod.643754 |
| 43 | 01.01.22.0168 | ASTA TEMPRATA AD INDUZIONE D20 L=980 |

**Figure 5.1**




| | | |
|---|---|---|
| 9 | 00.00.04.0352 **X2** | ANELLO ELASTICO UNI7437 PER FORO Ø32 |
| 5 | 00.00.04.0121 **X2** | MANICOTTO A RICIRCOLO DI SFERE LME-20UU |
| 32 | 01.01.21.0450 | MANICOTTO PORTA CUSCINETTI |

**Figure 5.2**


| | | |
|---|---|---|
| 14 | 00.00.04.0493 | VITE TCEI 8.8 M6X20 ZNT DIN 912 |
| 39 | 01.01.21.0459 | SUPPORTO CUSCINETTO SUPERIORE |
| 31 | 01.01.21.0434 | SUPPORTO BILANCIATORI ROTANTE |
| 7 | 00.00.04.0217 | CUSCINETTO 6905-2RS (25-42-9) |

**Figure 5.3**

# 6

# Conclusion

# 7
# Appendix A

## 7.1 Knowledge Base Examples

**Code in Atomese notation:**

Starting with the simplest arrangement of objects: there is a table and there is a screw, a bolt and a toolbox on it. Then, first the objects and their inheritances are defined, next the objects are differentiated, and finally, as they are all on top of the table, the *clear* status is associated with each of them.

```
1         (InheritanceLink
2                 (ConceptNode "screw")
3                 (ConceptNode "object"))
4         (InheritanceLink
5                 (ConceptNode "bolt")
6                 (ConceptNode "object"))
7         (InheritanceLink
8                 (ConceptNode "toolbox")
9                 (ConceptNode "object"))
10        (InheritanceLink
11                (ConceptNode "table")
12                (ConceptNode "fixed-object"))
13
```

```
14          (NotLink (EqualLink
15                  (ConceptNode "screw") (ConceptNode "bolt")))
16          (NotLink (EqualLink
17                  (ConceptNode "screw") (ConceptNode "toolbox")))
18          (NotLink (EqualLink
19                  (ConceptNode "screw") (ConceptNode "table")))
20          (NotLink (EqualLink
21                  (ConceptNode "bolt") (ConceptNode "toolbox")))
22          (NotLink (EqualLink
23                  (ConceptNode "bolt") (ConceptNode "table")))
24          (NotLink (EqualLink
25                  (ConceptNode "toolbox") (ConceptNode "table")))
26
27          (EvaluationLink
28                  (PredicateNode "clear")
29                  (ConceptNode "screw"))
30          (EvaluationLink
31                  (PredicateNode "clear")
32                  (ConceptNode "bolt"))
33          (EvaluationLink
34                  (PredicateNode "clear")
35                  (ConceptNode "toolbox"))
36          (EvaluationLink
37                  (PredicateNode "clear")
38                  (ConceptNode "table"))
```

Fixed objects do not lose their *clear* state when an object is placed on top of them, as is the case of the other objects. However, this state is maintained as long as there is room to place a new object, otherwise it is removed and the fixed object will result completely busy.
Furthermore, notice how there are no states related to the fact that the objects are on top of the table. This is because the actions of *Pickup* and *Putdown* have been maintained. The table is considered as the main working plane, so these actions will be executed within the objects above it.
In general, these actions can be used on objects that are on top of fixed objects in the initial situation, so for example, *Pickup* will not require to know what is under the object to pick up, it will just pick it up. *Putdown*, on the other hand, is used to put down an object in a

generic position, it is not important where. This action is rarely used.
However, all four actions are maintained and used to speed up the first actions and simplify the initial knowledge base.

To conclude, suppose now that the screw is on the toolbox and the bolt is in the hand. InheritanceLinks and NotLinks are the same as the code above and will always remain unchanged once created, within each execution of the algorithm, thus they are omitted below. The last part of the knowledge base will become as follows:

```
        (EvaluationLink
                (PredicateNode "clear")
                (ConceptNode "screw"))
        (EvaluationLink
                (PredicateNode "in-hand")
                (ConceptNode "bolt"))
        (EvaluationLink
                (PredicateNode "on")
                (ListLink
                        (ConceptNode "screw")
                        (ConceptNode "toolbox"))
        (EvaluationLink
                (PredicateNode "clear")
                (ConceptNode "table"))
```

## 7.2 PDDL Problem and Domain

**Code in PDDL notation:**

Description of the problem as a definition of the domain, objects, initial state (INIT) and goal.
In this example: four blocks (A, B, C, D) are arranged independently on the table and the robot's hand is free.
The goal is a stack formed by the four blocks placed in reverse alphabetical order (D on C on B on A).

```
(define (problem BLOCKS-4-0)
        (:domain BLOCKS)
        (:objects D B A C - block)
        (:INIT (CLEAR C) (CLEAR A) (CLEAR B) (CLEAR D)
```

```
                    (ONTABLE C) (ONTABLE A) (ONTABLE B)
                    (ONTABLE D) (HANDEMPTY))
          (:goal (AND (ON D C) (ON C B) (ON B A)))
)
```

**Code in PDDL notation:**

Description of the domain, concerning the problem above, as all possible actions and "predicates" available. Predicates are the "states" in which one or more objects can be. Actions, takeable by the robot, are described as input parameters and preconditions for performing that action and effects once the action is completed.

```
(define (domain BLOCKS)
  (:requirements :strips :typing)
  (:types block)
  (:predicates (on ?x - block ?y - block)
               (ontable ?x - block)
               (clear ?x - block)
               (handempty)
               (holding ?x - block)
               )

  (:action pick-up
             :parameters (?x - block)
             :precondition (and (clear ?x) (ontable ?x)
                                (handempty))
             :effect
             (and (not (ontable ?x))
                  (not (clear ?x))
                  (not (handempty))
                  (holding ?x)))

  (:action put-down
             :parameters (?x - block)
             :precondition (holding ?x)
             :effect
             (and (not (holding ?x))
                  (clear ?x)
```

```
                    (handempty)
                    (ontable ?x)))
  (:action stack
             :parameters (?x - block ?y - block)
             :precondition (and (holding ?x) (clear ?y))
             :effect
             (and (not (holding ?x))
                  (not (clear ?y))
                  (clear ?x)
                  (handempty)
                  (on ?x ?y)))
  (:action unstack
             :parameters (?x - block ?y - block)
             :precondition (and (on ?x ?y) (clear ?x)
                                (handempty))
             :effect
             (and (holding ?x)
                  (clear ?y)
                  (not (clear ?x))
                  (not (handempty))
                  (not (on ?x ?y)))))
```

## 7.3 Action Rules

**Code in Atomese notation:**

*Putdown* action rule. This rule works exactly in the same way as *Pickup*. The only differences are the preconditions and effects, which are reversed, and the absence of the "stacked" state atom, which is no longer required.

```
(define (putdown-action . args)
  (define precond (car (cdr args)))
  (define eff (car args))
  (cog-extract-recursive! precond)
  eff
)

(define (putdown args)
```

```
(let
  ((rule
    (if (equal? args '())
      (QueryLink
        (VariableList
          (TypedVariableLink
            (VariableNode "?ob")
            (TypeNode "ConceptNode"))
        ) ; parameters
        (PresentLink
          (InheritanceLink
            (VariableNode "?ob")
            (ConceptNode "object"))
          (EvaluationLink
            (PredicateNode "in_hand")
            (VariableNode "?ob"))
        )
        (ExecutionOutputLink
          (GroundedSchemaNode "scm: putdown-action")
          (ListLink
            ; effect
            (EvaluationLink
              (PredicateNode "clear")
              (VariableNode "?ob")
            )
            ; precondition
            (EvaluationLink
              (PredicateNode "in_hand")
              (VariableNode "?ob")
            )
          )
        )
      )
      (QueryLink
        (VariableList
          (TypedVariableLink
```

```
45              (VariableNode "?ob")
46              (TypeNode "ConceptNode"))
47          ) ; parameters
48          (AndLink
49            (PresentLink
50              (InheritanceLink
51                (VariableNode "?ob")
52                (ConceptNode "object"))
53              (EvaluationLink
54                (PredicateNode "in_hand")
55                (VariableNode "?ob"))
56            )
57            (EqualLink
58              (VariableNode "?ob")
59              args
60            )
61          )
62          (ExecutionOutputLink
63            (GroundedSchemaNode "scm: putdown-action")
64            (ListLink
65              ; effect
66              (EvaluationLink
67                (PredicateNode "clear")
68                (VariableNode "?ob")
69              )
70              ; precondition
71              (EvaluationLink
72                (PredicateNode "in_hand")
73                (VariableNode "?ob")
74              )
75            )
76          )
77        )
78      )
79    ))
80    rule
```

```
  )
)
```

**Code in Atomese notation:**

*Stack* action rule. This rule requires a more complex pattern than the first two, however, its structure and operation behave in the same way as the previous ones.

Firstly, two arguments are used to define which objects will form the stack. Also in the pattern, an OrLink type atom differentiates the case where a fixed object is coinvolved. Finally, the Scheme function *stack-action* must first extract the object above and the object below and then it can use them to create the effect atoms and remove the state atoms that are no longer valid.

```
(define (stack-action . args)
  (define precond (car (cdr args)))
  (define eff (car args))
  (define over_obj
    (cog-outgoing-atom (cog-outgoing-atom eff 1) 0))
  (define under_obj
    (cog-outgoing-atom (cog-outgoing-atom eff 1) 1))

  (cog-extract-recursive! precond)
  (cog-extract! (Predicate "in_hand"))

  (cog-extract! (Evaluation (Predicate "clear") under_obj))
  (cog-extract! (Predicate "clear"))

  (EvaluationLink (PredicateNode "clear") over_obj)
  (EvaluationLink (PredicateNode "stacked") over_obj)

  eff
)

(define (stack ob underob)
  (let
    ((rule
      (if (equal? ob '())
        (QueryLink
```

```
26            (VariableList
27              (TypedVariableLink
28                (VariableNode "?ob")
29                (TypeNode "ConceptNode"))
30              (TypedVariableLink
31                (VariableNode "?underob")
32                (TypeNode "ConceptNode"))
33            ) ; parameters
34            (AndLink
35              (PresentLink
36                (NotLink
37                  (EqualLink
38                    (VariableNode "?ob")
39                    (VariableNode "?underob")))
40                (InheritanceLink
41                  (VariableNode "?ob")
42                  (ConceptNode "object"))
43                (EvaluationLink
44                  (PredicateNode "in_hand")
45                  (VariableNode "?ob"))
46              )
47              (OrLink
48                (AndLink
49                  (InheritanceLink
50                    (VariableNode "?underob")
51                    (ConceptNode "object"))
52                  (EvaluationLink
53                    (PredicateNode "clear")
54                    (VariableNode "?underob"))
55                )
56                (AndLink
57                  (InheritanceLink
58                    (VariableNode "?underob")
59                    (ConceptNode "fixed-object"))
60                )
61              )
```

```
62            )
63            (ExecutionOutputLink
64              (GroundedSchemaNode "scm: stack-action")
65              (ListLink
66                ; effect
67                (EvaluationLink
68                  (PredicateNode "on")
69                  (ListLink
70                    (VariableNode "?ob")
71                    (VariableNode "?underob")
72                  )
73                )
74                ; precondition
75                (EvaluationLink
76                  (PredicateNode "in_hand")
77                  (VariableNode "?ob"))
78              )
79            )
80          )
81          (QueryLink
82            (VariableList
83              (TypedVariableLink
84                (VariableNode "?ob")
85                (TypeNode "ConceptNode"))
86              (TypedVariableLink
87                (VariableNode "?underob")
88                (TypeNode "ConceptNode"))
89            ) ; parameters
90            (AndLink
91              (PresentLink
92                (NotLink
93                  (EqualLink
94                    (VariableNode "?ob")
95                    (VariableNode "?underob")))
96                (InheritanceLink
97                  (VariableNode "?ob")
```

```
98                  (ConceptNode "object"))
99                (EvaluationLink
100                 (PredicateNode "in_hand")
101                 (VariableNode "?ob"))
102             )
103             (EqualLink
104               (VariableNode "?ob")
105               ob
106             )
107             (EqualLink
108               (VariableNode "?underob")
109               underob
110             )
111             (OrLink
112               (AndLink
113                 (InheritanceLink
114                   (VariableNode "?underob")
115                   (ConceptNode "object"))
116                 (EvaluationLink
117                   (PredicateNode "clear")
118                   (VariableNode "?underob"))
119               )
120               (AndLink
121                 (InheritanceLink
122                   (VariableNode "?underob")
123                   (ConceptNode "fixed-object"))
124               )
125             )
126           )
127           (ExecutionOutputLink
128             (GroundedSchemaNode "scm: stack-action")
129             (ListLink
130               ; effect
131               (EvaluationLink
132                 (PredicateNode "on")
133                 (ListLink
```

```
                    (VariableNode "?ob")
                    (VariableNode "?underob")
                  )
                )
                ; precondition
                (EvaluationLink
                  (PredicateNode "in_hand")
                  (VariableNode "?ob"))
              )
            )
          )
        )
      ))
      rule
  )
)
```

**Code in Atomese notation:**

*Unstack* action rule. Same considerations made for the rule of *Putdown*: this rule behaves like *Stack* with inverted preconditions and effects. While *Stack* creates the "stacked" state atom, *Unstack* removes it.

```
(define (unstack-action . args)
  (define precond (car (cdr args)))
  (define eff (car args))
  (define over_obj
    (cog-outgoing-atom (cog-outgoing-atom precond 1) 0))
  (define under_obj
    (cog-outgoing-atom (cog-outgoing-atom precond 1) 1))

  (cog-extract-recursive! precond)
  (cog-extract! (Predicate "on"))
  (cog-extract! (List over_obj under_obj))

  (cog-extract! (Evaluation (Predicate "clear") over_obj))
  (cog-extract! (Predicate "clear"))
  (cog-extract! (Evaluation (Predicate "stacked") over_obj))
  (cog-extract! (Predicate "stacked"))
```

```
17
18    (EvaluationLink (PredicateNode "clear") under_obj)
19
20    eff
21  )
22
23  (define (unstack ob underob)
24    (let
25      ((rule
26        (if (equal? ob '())
27          (QueryLink
28            (VariableList
29              (TypedVariableLink
30                (VariableNode "?ob")
31                (TypeNode "ConceptNode"))
32              (TypedVariableLink
33                (VariableNode "?underob")
34                (TypeNode "ConceptNode"))
35            ) ; parameters
36            (AndLink
37              (PresentLink
38                (NotLink
39                  (EqualLink
40                    (VariableNode "?ob")
41                    (VariableNode "?underob")))
42                (InheritanceLink
43                  (VariableNode "?ob")
44                  (ConceptNode "object"))
45                (EvaluationLink
46                  (PredicateNode "clear")
47                  (VariableNode "?ob"))
48                (EvaluationLink
49                  (PredicateNode "on")
50                  (ListLink
51                    (VariableNode "?ob")
52                    (VariableNode "?underob")
```

```
53                  )
54                )
55              )
56              (OrLink
57                (InheritanceLink
58                  (VariableNode "?underob")
59                  (ConceptNode "object"))
60                (InheritanceLink
61                  (VariableNode "?underob")
62                  (ConceptNode "fixed-object"))
63              )
64            )
65            (ExecutionOutputLink
66              (GroundedSchemaNode "scm: unstack-action")
67              (ListLink
68                ; effect
69                (EvaluationLink
70                  (PredicateNode "in_hand")
71                  (VariableNode "?ob"))
72                ; precondition
73                (EvaluationLink
74                  (PredicateNode "on")
75                  (ListLink
76                    (VariableNode "?ob")
77                    (VariableNode "?underob")
78                  )
79                )
80              )
81            )
82          )
83          (QueryLink
84            (VariableList
85              (TypedVariableLink
86                (VariableNode "?ob")
87                (TypeNode "ConceptNode"))
88              (TypedVariableLink
```

```
89                (VariableNode "?underob")
90                (TypeNode "ConceptNode"))
91            ) ; parameters
92            (AndLink
93              (PresentLink
94                (NotLink
95                  (EqualLink
96                    (VariableNode "?ob")
97                    (VariableNode "?underob")))
98                (InheritanceLink
99                  (VariableNode "?ob")
100                 (ConceptNode "object"))
101               (EvaluationLink
102                 (PredicateNode "clear")
103                 (VariableNode "?ob"))
104               (EvaluationLink
105                 (PredicateNode "on")
106                 (ListLink
107                   (VariableNode "?ob")
108                   (VariableNode "?underob")
109                 )
110               )
111             )
112             (OrLink
113               (InheritanceLink
114                 (VariableNode "?underob")
115                 (ConceptNode "object"))
116               (InheritanceLink
117                 (VariableNode "?underob")
118                 (ConceptNode "fixed-object"))
119             )
120             (EqualLink
121               (VariableNode "?ob")
122               ob
123             )
124             (EqualLink
```

```
                (VariableNode "?underob")
                underob
              )
            )
            (ExecutionOutputLink
              (GroundedSchemaNode "scm: unstack-action")
              (ListLink
                ; effect
                (EvaluationLink
                  (PredicateNode "in_hand")
                  (VariableNode "?ob"))
                ; precondition
                (EvaluationLink
                  (PredicateNode "on")
                  (ListLink
                    (VariableNode "?ob")
                    (VariableNode "?underob")
                  )
                )
              )
            )
          )
        )
      ))
      rule
  )
)
```

**Code in Atomese notation:**

This is an additional rule of utility: given an object as an argument, it returns the object immediately under it, if it exists.

It is used by the BFS-Based algorithm 4.5.1 when calling *Unstack* action.

```
(define (find_underob args)
  (QueryLink
    (VariableList
```

```
4         (TypedVariableLink
5           (VariableNode "?underob")
6           (TypeNode "ConceptNode"))
7       ) ; parameters
8       (AndLink
9         (PresentLink
10          (NotLink
11            (EqualLink args
12            (VariableNode "?underob")))
13          (EvaluationLink
14            (PredicateNode "on")
15              (ListLink
16                args
17                (VariableNode "?underob")
18              )
19          )
20        )
21        (OrLink
22          (InheritanceLink
23            (VariableNode "?underob")
24            (ConceptNode "object"))
25          (InheritanceLink
26            (VariableNode "?underob")
27            (ConceptNode "fixed-object"))
28        )
29      )
30      (VariableNode "?underob")
31    )
32  )
```

## 7.4 NLP Rules

**Code in Atomese notation:**

This rule infers all objects that are subject to a state but are independent of other objects. That is, the *clear* and *in-hand* state atoms.

```
(define find_goal
  (QueryLink
    (VariableList
      (TypedVariableLink
        (VariableNode "?status_id")
        (TypeNode "PredicateNode"))
      (TypedVariableLink
        (VariableNode "?status")
        (TypeNode "PredicateNode"))
      (TypedVariableLink
        (VariableNode "?ob_id")
        (TypeNode "ConceptNode"))
      (TypedVariableLink
        (VariableNode "?ob")
        (TypeNode "ConceptNode"))

    )
    (PresentLink
      (EvaluationLink
        (VariableNode "?status_id")
        (ListLink
          (VariableNode "?ob_id")
        )
      )
      (InheritanceLink
        (VariableNode "?ob_id")
        (VariableNode "?ob")
      )
      (ImplicationLink
        (VariableNode "?status_id")
        (VariableNode "?status")
      )
    )
    (EvaluationLink
      (VariableNode "?status")
      (VariableNode "?ob")
```

```
37        )
38     )
39  )
```

**Code in Atomese notation:**

This rule infers all objects subject to a stack. That is, the *on* state atoms.

However, the various EvaluationLink type atoms (lines 78-84) that one gets as results should already exist once Relex2Logic is run.

Unfortunately, it has errors which, instead of creating a single EvaluationLink that joins multiple ConceptNodes in the ListLink, create multiple EvaluationLinks with the same PredicateNode but separate for each object ( thus, the ListLinks contain only one ConceptNode each).

Since Relex2Logic is no longer maintained, this rule fixes this kind of error.

```
1  (define fix
2    (QueryLink
3      (VariableList
4        (TypedVariableLink
5          (VariableNode "?d1")
6          (TypeNode "DefinedLinguisticRelationshipNode"))
7        (TypedVariableLink
8          (VariableNode "?d2")
9          (TypeNode "DefinedLinguisticRelationshipNode"))
10       (TypedVariableLink
11         (VariableNode "?ob_w")
12         (TypeNode "WordInstanceNode"))
13       (TypedVariableLink
14         (VariableNode "?underob_w")
15         (TypeNode "WordInstanceNode"))
16       (TypedVariableLink
17         (VariableNode "?pred_w")
18         (TypeNode "WordInstanceNode"))
19       (TypedVariableLink
20         (VariableNode "?ob")
21         (TypeNode "ConceptNode"))
22       (TypedVariableLink
23         (VariableNode "?underob")
```

```
24          (TypeNode "ConceptNode"))
25        (TypedVariableLink
26          (VariableNode "?pred")
27          (TypeNode "PredicateNode"))
28        (TypedVariableLink
29          (VariableNode "?ob_id")
30          (TypeNode "ConceptNode"))
31        (TypedVariableLink
32          (VariableNode "?underob_id")
33          (TypeNode "ConceptNode"))
34        (TypedVariableLink
35          (VariableNode "?pred_id")
36          (TypeNode "PredicateNode"))
37      )
38      (PresentLink
39        (EvaluationLink
40          (VariableNode "?d1")
41          (ListLink
42            (VariableNode "?pred_w")
43            (VariableNode "?underob_w")
44          )
45        )
46        (EvaluationLink
47          (VariableNode "?d2")
48          (ListLink
49            (VariableNode "?ob_w")
50            (VariableNode "?pred_w")
51          )
52        )
53        (InheritanceLink
54          (VariableNode "?ob_id")
55          (VariableNode "?ob")
56        )
57        (InheritanceLink
58          (VariableNode "?underob_id")
59          (VariableNode "?underob")
```

```
60          )
61          (ImplicationLink
62            (VariableNode "?pred_id")
63            (VariableNode "?pred")
64          )
65          (ReferenceLink
66            (VariableNode "?ob_id")
67            (VariableNode "?ob_w")
68          )
69          (ReferenceLink
70            (VariableNode "?underob_id")
71            (VariableNode "?underob_w")
72          )
73          (ReferenceLink
74            (VariableNode "?pred_id")
75            (VariableNode "?pred_w")
76          )
77        )
78        (EvaluationLink
79          (VariableNode "?pred")
80          (ListLink
81            (VariableNode "?ob")
82            (VariableNode "?underob")
83          )
84        )
85      )
86  )
```

# Bibliography


[1] Leah Davidson. *Narrow vs. general ai: What's next for artificial intelligence?* Aug. 2019. URL: https://www.springboard.com/blog/ai-machine-learning/narrow-vs-general-ai/.

[2] *The three different types of artificial intelligence – ANI, AGI AND ASI*. Oct. 2019. URL: https://www.ediweekly.com/the-three-different-types-of-artificial-intelligence-ani-agi-and-asi/.

[3] Lawtomated. *AI for Legal: Ani, AGI and ASI*. Feb. 2020. URL: https://lawtomated.medium.com/ai-for-legal-ani-agi-and-asi-cea6e7a4079e.

[4] Sean Allan. *The three tiers of AI: Automating tomorrow with Agi, ASI and ANI*. Sept. 2018. URL: https://www.aware.co.th/three-tiers-ai-automating-tomorrow-agi-asi-ani/.

[5] Astute Solutions. *Artificial narrow intelligence and the customer experience*. Nov. 2020. URL: https://astutesolutions.com/ani/artificial-narrow-intelligence.

[6] Richa Bhatia. *AGI vs ANI and understanding the path towards machine intelligence*. Jan. 2018. URL: https://analyticsindiamag.com/agi-vs-ani-understanding-the-path-towards-machine-intelligence/.

[7] Ben Goertzel. *Artificial general intelligence*. URL: https://wiki.opencog.org/w/Artificial_General_Intelligence#:~:text=types%20of%20problems.-,Overview%20of%20AGI,-edit.

[8] Cem Dilmegani. *When will singularity happen? 995 experts' opinions on AGI*. Sept. 2021. URL: https://research.aimultiple.com/artificial-general-intelligence-singularity-timing/.

[9] Mark Chen et al. "Evaluating Large Language Models Trained on Code". In: *CoRR* abs/2107.03374 (2021). arXiv: 2107.03374. URL: https://arxiv.org/abs/2107.03374.

[10] Alec Radford et al. “Learning Transferable Visual Models From Natural Language Supervision”. In: *CoRR* abs/2103.00020 (2021). arXiv: 2103.00020. URL: https://arxiv.org/abs/2103.00020.

[11] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

[12] Alberto Romero. *GPT-3 - a complete overview*. May 2021. URL: https://towardsdatascience.com/gpt-3-a-complete-overview-190232eb25fd.

[13] V. Vemuri. “Main problems and issues in neural networks application”. In: *1993 First New Zealand International Two-Stream Conference on Artificial Neural Networks and Expert Systems*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 1993, p. 226. DOI: 10.1109/ANNES.1993.323037. URL: https://doi.ieeecomputersociety.org/10.1109/ANNES.1993.323037.

[14] Christian Szegedy et al. “Intriguing properties of neural networks”. In: *International Conference on Learning Representations*. 2014. URL: http://arxiv.org/abs/1312.6199.

[15] Adam Gleave et al. “Adversarial Policies: Attacking Deep Reinforcement Learning”. In: *CoRR* abs/1905.10615 (2019). arXiv: 1905.10615. URL: http://arxiv.org/abs/1905.10615.

[16] Nicolas Papernot et al. “The Limitations of Deep Learning in Adversarial Settings”. In: *2016 IEEE European Symposium on Security and Privacy (EuroS P)*. 2016, pp. 372–387. DOI: 10.1109/EuroSP.2016.36.

[17] Ian Goodfellow. *Attacking machine learning with adversarial examples*. Oct. 2020. URL: https://openai.com/blog/adversarial-example-research/.

[18] Ben Dickson. *The untold story of gpt-3 is the transformation of openai*. Sept. 2020. URL: https://bdtechtalks.com/2020/08/17/openai-gpt-3-commercial-ai/.

[19] Kyle Wiggers. *OpenAI’s massive gpt-3 model is impressive, but size isn’t everything*. May 2021. URL: https://venturebeat.com/2020/06/01/ai-machine-learning-openai-gpt-3-size-isnt-everything/.

[20] Vanessa Buhrmester, David Münch, and Michael Arens. “Analysis of Explainers of Black Box Deep Neural Networks for Computer Vision: A Survey”. In: *CoRR* abs/1911.12116 (2019). arXiv: 1911.12116. URL: http://arxiv.org/abs/1911.12116.

[21] *Backpropagation networks*. URL: http://slideplayer.com/slide/6816119.

[22] Niklas Donges. *4 reasons why deep learning and neural networks aren't always the right choice*. URL: https://builtin.com/data-science/disadvantages-neural-networks.

[23] Yuan Gong and Christian Poellabauer. "An Overview of Vulnerabilities of Voice Controlled Systems". In: Mar. 2018.

[24] Ben Goertzel. "Cognitive synergy: A universal principle for feasible general intelligence". In: June 2009, pp. 464–468. DOI: 10.1109/COGINF.2009.5250694.

[25] Ben Goertzel. *The hidden pattern : a patternist philosophy of mind*. Boca Raton: BrownWalker Press, 2006. ISBN: 1581129890.

[26] Ben Goertzel, Cassio Pennachin, and Nil Geisweiller. *Engineering General Intelligence, Part 1 - A Path to Advanced AGI via Embodied Learning and Cognitive Synergy*. Vol. 5. Atlantis Thinking Machines. Atlantis Press, 2014. ISBN: 978-94-6239-026-3. DOI: 10.2991/978-94-6239-027-0. URL: https://doi.org/10.2991/978-94-6239-027-0.

[27] Ben Goertzel, Cassio Pennachin, and Nil Geisweiller. *Engineering General Intelligence, Part 2 - The CogPrime Architecture for Integrative, Embodied AGI*. Vol. 6. Atlantis Thinking Machines. Atlantis Press, 2014. ISBN: 978-94-6239-029-4. DOI: 10.2991/978-94-6239-030-0. URL: https://doi.org/10.2991/978-94-6239-030-0.

[28] Linas Vepstas. *Graphs, metagraphs, ram, cpu*. 2020. URL: https://wiki.opencog.org/w/File:Ram-cpu.pdf.

[29] Amit Basu and Robert Blanning. *Metagraphs and Their Applications*. Jan. 2007. ISBN: 978-0-387-37233-4. DOI: 10.1007/978-0-387-37234-1.

[30] Ben Goertzel. "Folding and Unfolding on Metagraphs". In: *CoRR* abs/2012.01759 (2020). arXiv: 2012.01759. URL: https://arxiv.org/abs/2012.01759.

[31] Ben Goertzel. *Probabilistic logic networks : a comprehensive framework for uncertain inference*. New York, NY: Springer, 2008. ISBN: 978-1441945785.

[32] B. Goertzel et al. "Real-World Reasoning: Toward Scalable, Uncertain Spatiotemporal, Contextual and Causal Inference". In: *Atlantis Thinking Machines*. 2011.

[33] Ben Goertzel et al. July 2020. URL: https://wiki.opencog.org/wikihome/images/7/77/OpenCog_Hyperon.pdf.

[34] Ben Goertzel. "What Kind of Programming Language Best Suits Integrative AGI?" In: *CoRR* abs/2004.05267 (2020). arXiv: 2004.05267. URL: https://arxiv.org/abs/2004.05267.

[35] July 2017. URL: https://wiki.opencog.org/wikihome/images/5/53/Distributed_Atom_Space_-_Requirements_and_Ideas_-_DRAFT.pdf.

[36] André Senna. Aug. 2018. URL: https://wiki.opencog.org/wikihome/images/f/fd/Distributed_Atom_Space_-_Architecture.pdf.

[37] Alexey Potapov. Mar. 2020. URL: https://wiki.opencog.org/wikihome/images/0/0d/Distributed_Atomspace_2.0_and_graph_DBs_differences.pdf.

[38] Alexey Potapov. Apr. 2020. URL: https://wiki.opencog.org/wikihome/images/a/ab/Pattern_Mining%2C_Indexing_and_Generalized_Hypergraph_Representation.pdf.

[39] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. DOI: 10.1017/CBO9781139172752.

[40] Nil Geisweiller. *Unified rule engine*. Feb. 2019. URL: https://wiki.opencog.org/w/Unified_rule_engine.

[41] Edwin Olson. "AprilTag: A robust and flexible visual fiducial system". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2011, pp. 3400–3407.

[42] John Wang and Edwin Olson. "AprilTag 2: Efficient and robust fiducial detection". In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Oct. 2016.

[43] *Breadth-first search*. Sept. 2021. URL: https://en.wikipedia.org/wiki/Breadth-first_search.

[44] Jason Holdsworth. "The Nature of Breadth-First Search". In: (Feb. 1999).