B5IM2034

Master's Thesis

Semantic parsing with a domain ontology
領域オントロジーを用いた セマンティックパージング

Corentin DUMONT

March 24, 2017

Graduate School of Information Sciences
Tohoku University

A Master's Thesis
submitted to System Information Sciences,
Graduate School of Information Sciences,
Tohoku University
in partial fulfillment of the requirements for the degree of
Master of Information Sciences

Corentin DUMONT

Thesis Committee:

Professor Kentaro Inui	(Supervisor)
Professor Yoshifumi Kitamura	
Professor Tetsuo Kinoshita	
Associate Professor Naoaki Okazaki	(Co-supervisor)

Semantic parsing with a domain ontology
領域オントロジーを用いた セマンティックパージング[*]

Corentin DUMONT

Abstract

Using restricted domains in question answering (QA) allows to build an ontology of the domain that can be used with by a semantic parser to improve the performance of a QA system. However, building an ontology and a semantic parser can be costly and usually require a large amount of annotated data that may not be available for specific domains. Our work is a case study on how to build a semantic parser and a domain ontology with very low resource for the video game Minecraft.
We show that the creation of a semantic parser and the ontology it is based on can be done with a very low amount of expert manual annotations, facilitated by the use of automatically collected data from the web, and that fairly good performance can be obtained. Thus, as a contribution, this case study is a first step in solving the common problem of building a QA system in a restricted domain from very low resource and low manual effort.

Keywords:

Knowledge acquisition, Ontology, Semantic parsing, Question Answering

[*]Master's Thesis, System Information Sciences, Graduate School of Information Sciences, Tohoku University, B5IM2034, March 24, 2017.

Contents

List of Figures

List of Tables

1 Introduction

The initial motivation of this work is to improve the interaction between the players and Non-Player Characters (NPCs) in video games. As in Figure 1, in current video games, interaction between the player and the NPCs is very limited in the sense that the player cannot ask his own questions. Indeed, the game will often propose several possible questions or choices to the player, who can only choose among them.

Our final goal is to develop a natural language question answering system that

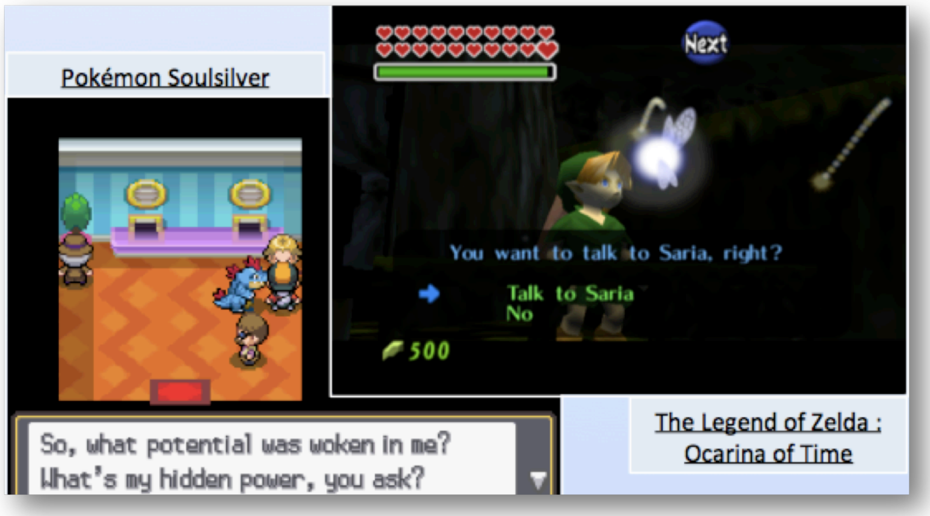

Pokémon Soulsilver
So, what potential was woken in me?
What's my hidden power, you ask?
Next
You want to talk to Saria, right?
Talk to Saria
No
500
The Legend of Zelda : Ocarina of Time

Figure 1: Interaction between players and NPCs in current video games

would make possible a real interaction between the player and the NPCs. Such a system, as shown in Figure 2, would allow the player to ask about anything in English about the content of the game, and make the NPCs able to answer in natural English to any question of the player as long as it is about the content of the game.

Unlike many QA systems that are designed to answer real world questions [1, 2], the final goal of this research is to build a system that can answer questions using the logic specific to the game, which may not be identical to the logic in the real world. For our study, we choose a popular game called Minecraft, whose openness provides a great liberty for players, which guarantees a large number of possible

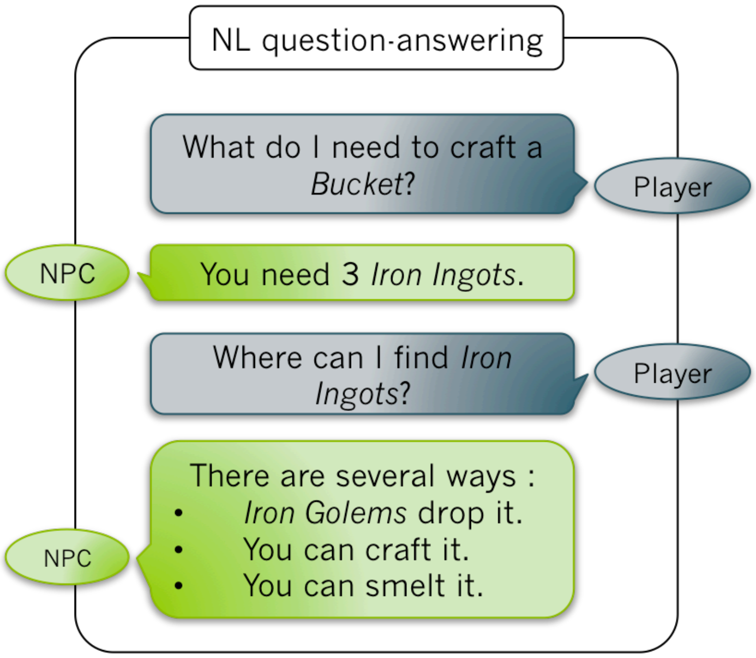


Figure 2: The final goal of our work, a natural language QA system

questions to ask about the game, and yet the presence of a specific logic that limits the actions of players. We are interested in this problem setting because it provides a testbed for combining Natural Language Processing with advanced logical inference techniques.

Our final system should be able to access knowledge about the content of the game, so we need to collect this knowledge. It should also understand the content of this knowledge and the players' questions in order to construct a relevant answer, so we need to build a semantic parser that will extract the meaning from English sentences by finding the important instances and the relations that link these instances together. Finally, it has to answer the questions by constructing relevant answers from the knowledge it has about the game using logical inference techniques.

As a whole, our work is a case study on how to build a Semantic Parser (and in the end, a complete Question Answering system) from very low resource in the restricted domain of a video game.

2 Case study: Minecraft

2.1 Minecraft

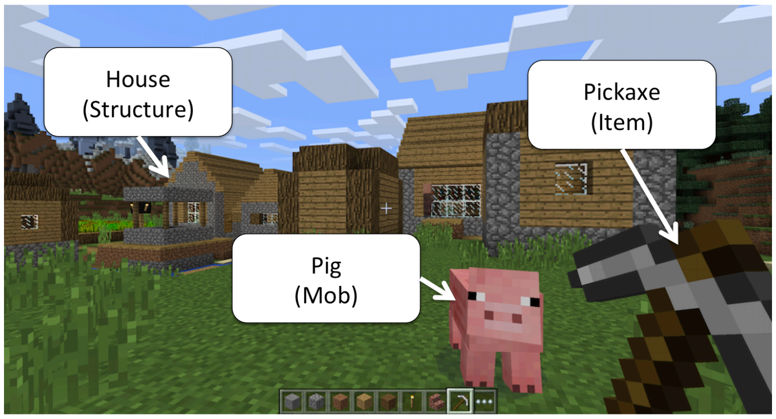
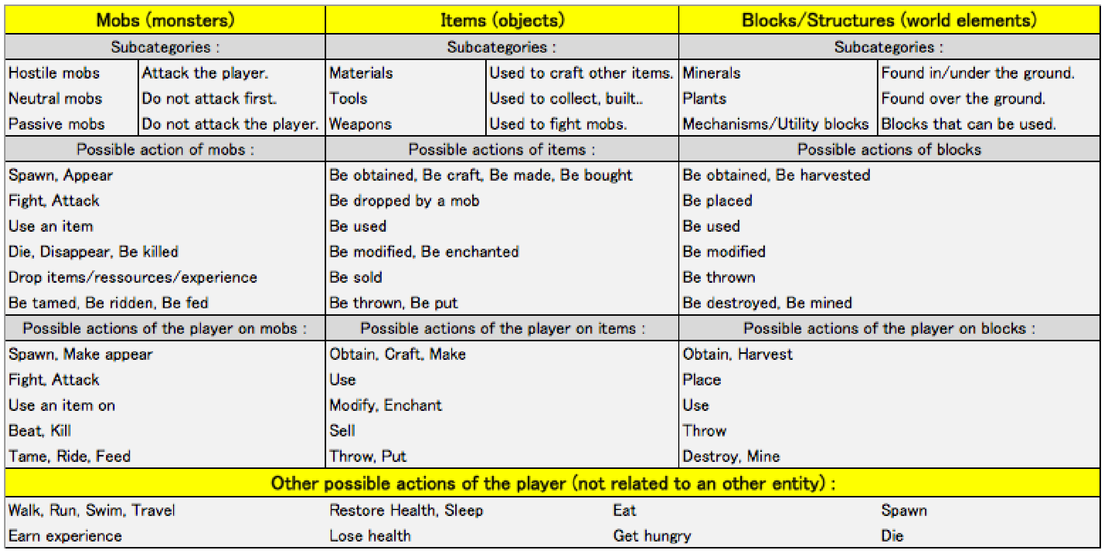


Figure 3: A snapshot of Minecraft



Mobs (monsters)		Items (objects)		Blocks/Structures (world elements)	
Subcategories :		Subcategories :		Subcategories :	
Hostile mobs	Attack the player.	Materials	Used to craft other items.	Minerals	Found in/under the ground.
Neutral mobs	Do not attack first.	Tools	Used to collect, built..	Plants	Found over the ground.
Passive mobs	Do not attack the player.	Weapons	Used to fight mobs.	Mechanisms/Utility blocks	Blocks that can be used.
Possible action of mobs :		Possible actions of items :		Possible actions of blocks	
Spawn, Appear		Be obtained, Be craft, Be made, Be bought		Be obtained, Be harvested	
Fight, Attack		Be dropped by a mob		Be placed	
Use an item		Be used		Be used	
Die, Disappear, Be killed		Be modified, Be enchanted		Be modified	
Drop items/ressources/experience		Be sold		Be thrown	
Be tamed, Be ridden, Be fed		Be thrown, Be put		Be destroyed, Be mined	
Possible actions of the player on mobs :		Possible actions of the player on items :		Possible actions of the player on blocks :	
Spawn, Make appear		Obtain, Craft, Make		Obtain, Harvest	
Fight, Attack		Use		Place	
Use an item on		Modify, Enchant		Use	
Beat, Kill		Sell		Throw	
Tame, Ride, Feed		Throw, Put		Destroy, Mine	
Other possible actions of the player (not related to an other entity) :					
Walk, Run, Swim, Travel		Restore Health, Sleep	Eat		Spawn
Earn experience		Lose health	Get hungry		Die

Figure 4: Entity types and actions in Minecraft

Minecraft (Figure 3) is a sandbox video game, which means that the player is free to choose the actions he wants to execute, and the order of these actions. However, as all video games, the number of possible actions is limited. The

main occupation of the player in Minecraft is to survive in a world populated by monsters, by finding resources (e.g. mining minerals, growing plants, etc.), to create structures, items and weapons (i.e. crafting them with the collected resources by following recipes) and beating monsters using crafted weapons and items to protect the created structures and earn experience and new items, in order to continue to develop.
The different entities of the game follow a hierarchy. Entities can be divided into Structures, Mobs (monsters), and Objects, which are divided into Items (for example used to fight) and Blocks (used to build Structures), and so on. The player can interact with these entities, or make independent actions. These actions and interactions are summarized in the Figure 4.
Minecraft players usually ask about:
properties of entities,

> *How many health point does a iron golem have?* (100)

relations between entities,

> *What is a cake composed of?* (milk, sugar, egg and wheat)

or conditions and effects of actions.

> *What pickaxe is needed to break a diamond block?* (iron pickaxe or diamond pickaxe)
> *How can I get a door?* (craft door, find door in village)

The knowledge needed to answer most of Minecraft players' questions are available on several websites dedicated to Minecraft that are often written by players themselves.

2.2 Why Minecraft?

Why did we choose Minecraft for our case study?
The liberty of the player guarantees a large number of possible questions to ask about the game, but the game follows nonetheless a logic that can be learned by a QA system to increase its ability to understand the meaning of the questions. In other words, Minecraft players are likely to ask non-trivial and difficult questions like:

What is the best way to obtain Obsidian?

To answer this question, we need to know about all the different ways to obtain *obsidian*, and we need to compare these methods, which is not possible if we do not understand them. However, the strict logic that rules the game makes these difficult questions possible to answer by an automatic QA system, contrary to such questions in the real world. For example, thanks to this strict logic, in the following sentences:

The player kills a wolf with a ???.

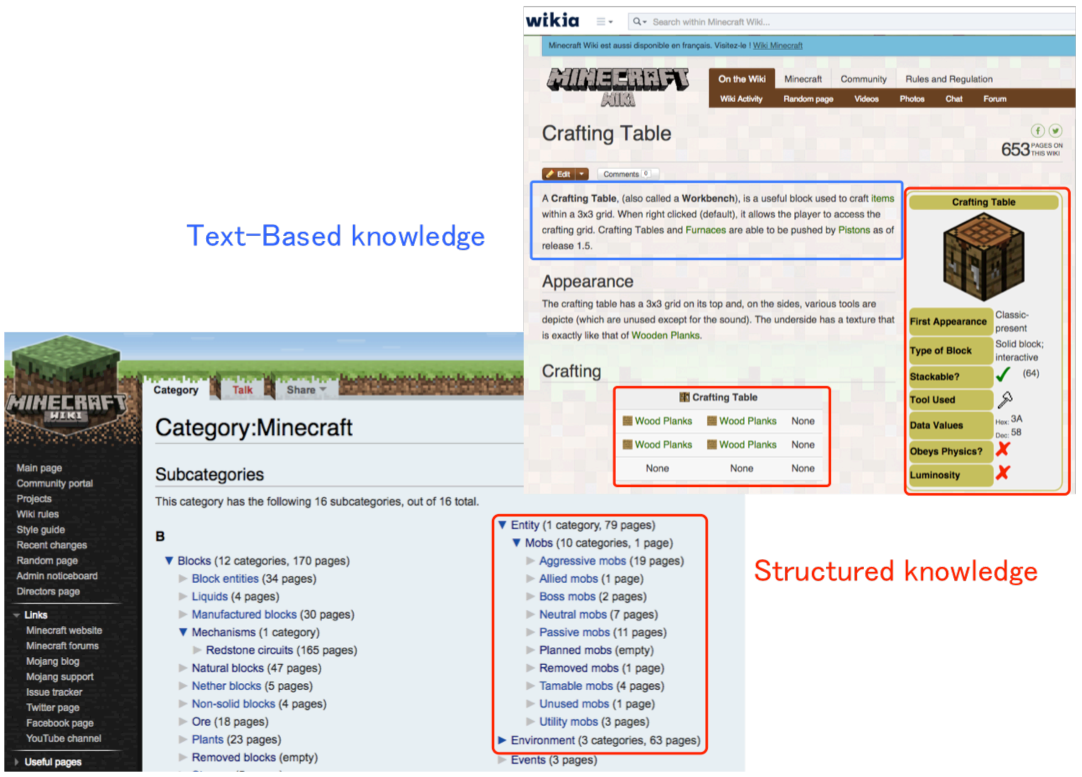
an automatic system can deduce that ??? can only be replaced with *sword*, *pickaxe*, *axe* or *shovel*. This kind of logic can be used to resolve ambiguities.
Our final goal is to build a QA system that can translate a question asked by players to a series of queries about the entities and actions in the game, and find a relevant answer in the knowledge database among all the information about these entities and actions, possibly by combining several pieces of knowledge together to answer difficult questions.
Since Minecraft is a popular game, we expect that we can find abundant data from the Web. Furthermore, Minecraft has been the domain for other works on artificial intelligence [3, 4], and not only in natural language processing (NLP). So choosing Minecraft as a domain can make our work more valuable and have a better impact.
There have been many efforts done to restrict the domain of the QA task and pursue some advanced reasoning. The Todai Robot Project [5] restricts the domain to university entrance exam questions. Other research includes solving algebra word problems [6] and instructing robots [7]. As a complement to these previous works, we believe the using of an open world video game as the domain has several merits. Firstly, the logic in a video game is simpler than the real world, which means that it can be handled readily. Therefore, this domain may provide a convenient testbed for integrating logical inference techniques into NLP systems, such as the logical inference using dependency-based compositional semantics [8]. Secondly, despite the rather simple rules, open world video games provide enough liberty for players, and their popularity attracts people to ask many questions about them, including creative and fun questions that can be

solved only by completely understanding the rules and logically combining them. Therefore, we expect the domain to be interesting and challenging as well.

3 Collect data about Minecraft

3.1 Knowledge corpus

A knowledge database is extracted from three different wiki-like websites[1]. These websites are constituted of pages describing an entity or a concept of the game. Similar to those from Wikipedia, the web-pages can be divided into two parts, namely the structured data such as infoboxes, tables, or web-pages hierarchy, and unstructured data such as natural language texts (Figure 5).



Crafting Table

A **Crafting Table**, (also called a **Workbench**), is a useful block used to craft items within a 3x3 grid. When right clicked (default), it allows the player to access the crafting grid. Crafting Tables and Furnaces are able to be pushed by Pistons as of release 1.5.

Appearance

The crafting table has a 3x3 grid on its top and, on the sides, various tools are depicte (which are unused except for the sound). The underside has a texture that is exactly like that of Wooden Planks.

Crafting

Crafting Table		
Wood Planks	Wood Planks	None
Wood Planks	Wood Planks	None
None	None	None

Crafting Table	
First Appearance	Classic-present
Type of Block	Solid block; interactive
Stackable?	✓ (64)
Tool Used	
Data Values	Hex: 3A Dec: 58
Obeys Physics?	✗
Luminosity	✗

Category:Minecraft

Subcategories

This category has the following 16 subcategories, out of 16 total.

B

- ▼ Blocks (12 categories, 170 pages)
  - ▶ Block entities (34 pages)
  - ▶ Liquids (4 pages)
  - ▶ Manufactured blocks (30 pages)
  - ▼ Mechanisms (1 category)
    - ▶ Redstone circuits (165 pages)
  - ▶ Natural blocks (47 pages)
  - ▶ Nether blocks (5 pages)
  - ▶ Non-solid blocks (4 pages)
  - ▶ Ore (18 pages)
  - ▶ Plants (23 pages)
  - ▶ Removed blocks (empty)

- ▼ Entity (1 category, 79 pages)
  - ▼ Mobs (10 categories, 1 page)
    - ▶ Aggressive mobs (19 pages)
    - ▶ Allied mobs (1 page)
    - ▶ Boss mobs (2 pages)
    - ▶ Neutral mobs (7 pages)
    - ▶ Passive mobs (11 pages)
    - ▶ Planned mobs (empty)
    - ▶ Removed mobs (1 page)
    - ▶ Tamable mobs (4 pages)
    - ▶ Unused mobs (1 page)
    - ▶ Utility mobs (3 pages)
- ▶ Environment (3 categories, 63 pages)
- ▶ Events (3 pages)

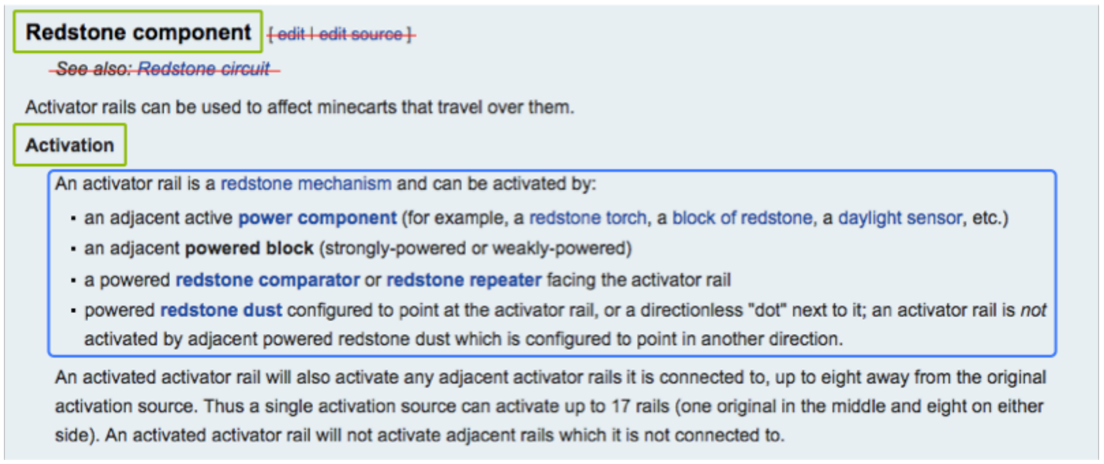
Figure 5: Text and structured data in the websites used for knowledge extraction

We preserve the structures of infoboxes and tables in our extraction, and separate structured and unstructured data. As a result, we obtain a database com-

[1] www.minecraft.gamepedia.com
www.minecraft.wikia.com
www.minecraftguides.org

posed of 1063 unique text files, organized in 51 folders and sub-folders to regroup related objects (see Table 1 for details).

To improve the quality of the extracted data, we defined extracting rules for each of the 3 websites. The main purpose of these rules is to extract separately structured and unstructured data, to remove useless information and to avoid extracting segments of sentences that loose their meaning out of context (see Figure 6)



Redstone component ~~[edit | edit source]~~

~~See also: Redstone circuit~~

Activator rails can be used to affect minecarts that travel over them.

Activation

An activator rail is a redstone mechanism and can be activated by:

- an adjacent active **power component** (for example, a redstone torch, a block of redstone, a daylight sensor, etc.)
- an adjacent **powered block** (strongly-powered or weakly-powered)
- a powered **redstone comparator** or **redstone repeater** facing the activator rail
- powered **redstone dust** configured to point at the activator rail, or a directionless "dot" next to it; an activator rail is *not* activated by adjacent powered redstone dust which is configured to point in another direction.

An activated activator rail will also activate any adjacent activator rails it is connected to, up to eight away from the original activation source. Thus a single activation source can activate up to 17 rails (one original in the middle and eight on either side). An activated activator rail will not activate adjacent rails which it is not connected to.

Figure 6: Extraction of Minecraft knowledge from websites

3.2 Question corpus

A corpus of questions and answers has been created on the basis of posts extracted from quiz websites[2]. 754 questions have been collected from different websites, and 544 of them have been extracted with their answer.

Then, we manually selected 100 relevant questions. For our purpose, it is important that the questions deal with facts inside the game, and not facts that are outside the game (e.g. questions about the creator of the game, the programming language used, etc.).

[2] www.quizlet.com
www.allthetests.com
www.gamefaqs.com

Folders		Number of Files	Description
DB-Gamepedia (from minecraft.gamepedia.com)	Blocks	154	Environment's blocks
	Entity	72	Mobs (Monsters)
	Items	161	Objects used by the player
	Others	183	Gameplay, History, etc.
	Total: 530 files		
DB-Wikia (from minecraft.wikia.com)	Blocks	200	
	Items	167	
	Main	23	Important objects/entities
	Mobs	56	
	Total: 392 files		
DB-Guides (from minecraftguides.org)	Blocks	101	Minerals, Plants, etc.
	Brewing	34	Recipes of potions
	Building	7	
	Farming	7	
	Items	77	Food, Tools, Weapons, etc.
	Main	10	Summaries of sub-folders
	Mini-Games	25	
	Mobs	27	
	Tutorials	30	
	Total: 300 files		
Total: 1063 unique files (1222 extracted files)			

Table 1: A summary of the knowledge database

Factoid questions:
What Item should I use to tame a Wolf?
Are Spiders Hostile?
Non-factoid questions:
What is the best way to spawn the two different types of Golem?
Is it interesting to kill the Ender Dragon?

Table 2: Example of factoid and non-factoid questions

We selected both factoid and non-factoid questions (see Table 2), and tried to include as much language variety (vocabulary and grammar) as possible in the reference to the concepts of the game. For each selected question, we wrote about nine questions with the same meaning but asked differently, or with a close or related meaning (Table 3).

0	*Where do you find a Mushroom?*
1	*How do you obtain a Mushroom?*
2	*How do I get a Mushroom?*
3	*Where can I get a Mushroom?*
4	*Where can I obtain a Mushroom?*
5	*What is a way to get a Mushroom?*
6	*How to get Mushrooms?*
7	*Where do I find Mushrooms?*
8	*Where are Mushrooms located?*
9	*Where can Mushrooms be found?*

Table 3: Examples of similar questions written

This way, we obtained a corpus that can be used for handling language variations in the QA task. We finally obtained a corpus of 1684 questions, among which 928 has been written on the basis of 100 relevant questions.

3.2.1 Can the questions be answered by the knowledge corpus?

From the questions that have been extracted from quizzes websites, we can distinguish 3 types of questions. Some questions are not relevant, because they deal with some facts external to the game itself, or because they contain a mistake:

> *What is the name of the famous yellow duck who plays Minecraft on YouTube?* (YouTube is external to the game.)
> *What are the 5 types of wood?* (There are actually 6 types of wood.)

Some questions can be "easily" answered with the knowledge database. This is the case when the answer is clearly written in the database (for example a numeric value in a table). The questions that can be "easily" answered are often factoid questions:

> *How many hearts does a Giant have?* (The answer, 50 hearts, is written in the infobox of the Giant.)

This kind of question can be answered simply by locating the place where the answer is written in the database. However, some questions can be answered only by computing the answer using crossed information. This is the case of non-factoid questions, which can be considered as "difficult" to answer:

> *What is the best strategy for finding diamonds?* (To answer this question, the system has to find all the different ways to find diamonds and evaluate the efficiency of each method. This evaluation is challenging because the criteria for a good strategy are not stated in the question.)

In our data, the non-relevant questions are rare (about 2%), whereas the non-factoid questions are quite common (about 20%), which provides a good motivation for a QA system to handle complicated questions.
However, answering non-factoid questions implies reasoning, i.e. logic inference on different pieces of information contained in the knowledge database, which can only be done on a structured database. This is why we defined the ontology of a meaning representation, which is used to structure the natural sentences of the database to their logical form. The challenge is to balance the complexity

of the meaning representation, as it must be simple to take advantage of the simple logic of the game, but must be expressive enough so that it can be used to structure all the useful information.

3.3 Minecraft Ontology

Our ontology defines two types of classes to represent the instances in Minecraft and the relations that links these instances together.

The **Instance classes** regroup Minecraft Entities and Events (Figure 7).

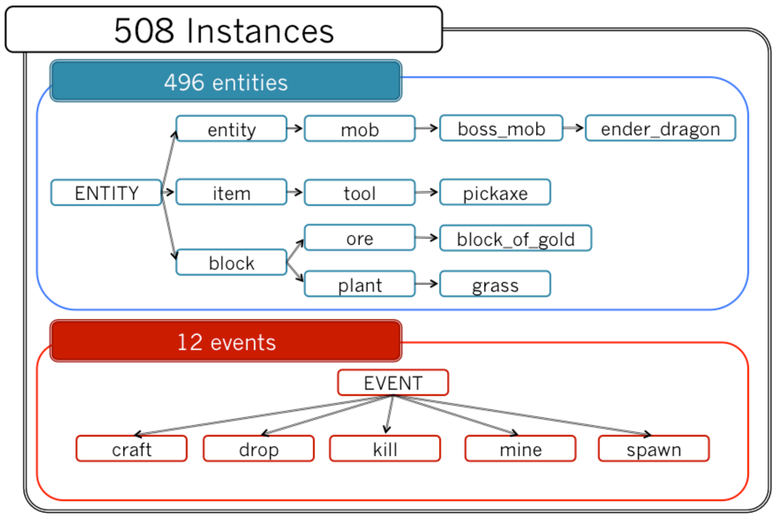


Figure 7: Instances of the Ontology

Entity classes include all Structures, Objects (e.g. Blocks, Items, *etc.*) and Mobs that the player can interact with. The ontology defines the list of 496 Entities and their hierarchy (e.g. the Minecraft Entity "*Stone*" is a subclass of "*Natural Block*"). The list of entities and their hierarchy is constructed by checking named entities appeared in our QA corpus and using the hierarchy of web-pages of the Gamepedia website. This list is supposed to have high coverage. We have regrouped some concepts that are usually used by players as different ones but are actually the same objects in the game. For example, both "*chicken*" and "*chick*" are represented by the same Minecraft Entity class *Chicken*, and are considered as two variations of this class. The regrouping is done because these entities have similar interactions with the player and other entities, and has been mainly done automatically by using the infoboxes of the Gamepedia website that lists the possible variations of entities.

Event classes are used to represent events or actions in the game. We make the list of Event classes by considering possible operations by the player and checking questions asked in our QA corpus. We tried to minimize the number of Event classes by regrouping some events that can be expressed as the same actions linked to different Minecraft Entities. For example, both the actions *sleep* and *eat* are regrouped into the event *use*, because "*sleeping*" and "*eating*" are equivalent to "*using*" the Minecraft Entities *bed* and *food*, respectively. There is currently no hierarchy between event classes, except a common root EVENT class, but the fact that entities and events are regrouped under instance classes and used similarly by our classification models would make very easy to add an event hierarchy in the ontology.

Besides the hierarchy, each instance class is further defined in the ontology (example in Figure 8).

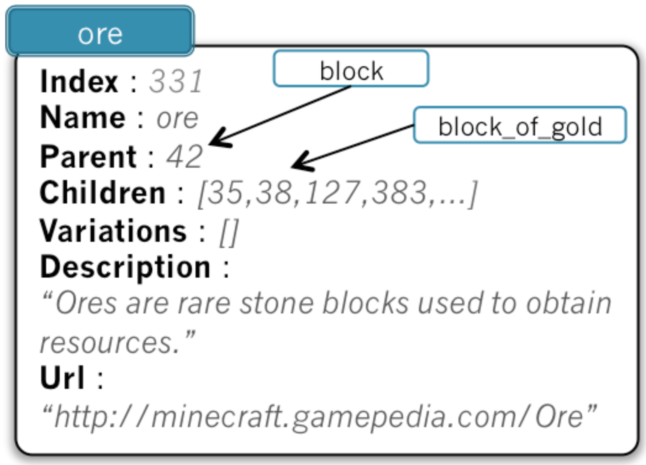

ore
Index : 331
Name : ore
Parent : 42
Children : [35,38,127,383,...]
Variations : []
Description :
"Ores are rare stone blocks used to obtain resources."
Url :
"http://minecraft.gamepedia.com/Ore"
block
block_of_gold

Figure 8: Example of instance: Ore (entity)

The **Relation classes** represent a link between two instances. The ontology defines the list of 54 relation classes and the constraints that define what instances classes can be linked using each relation class. Relation classes are divided in assertion relations (or factual relations, we will use "_" at the beginning of the name) and non-assertion relations (see Figure 9). While assertion relations are used to express a piece of information (e.g. property of an instance, or causality link between to events), non-assertion relations are used to express the semantic link between two instances that appear in the same piece of information (e.g. in the sentence "The player can kill a wolf with a golden sword", the "golden sword" is the weapon used to "kill", so the event "kill" is linked to the entity "golden sword" by the relation "weapon".)

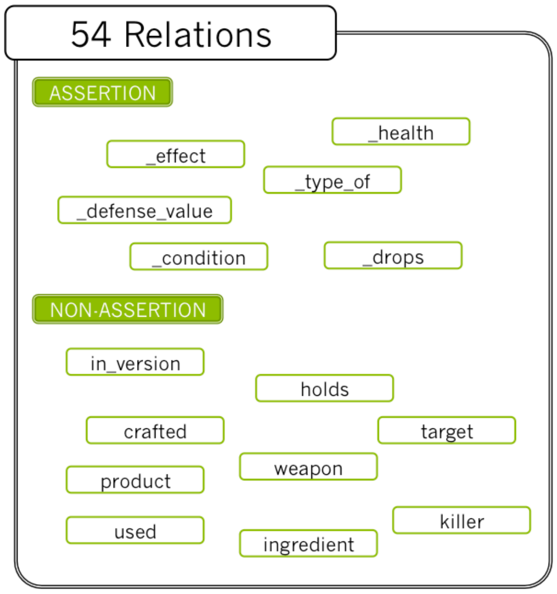


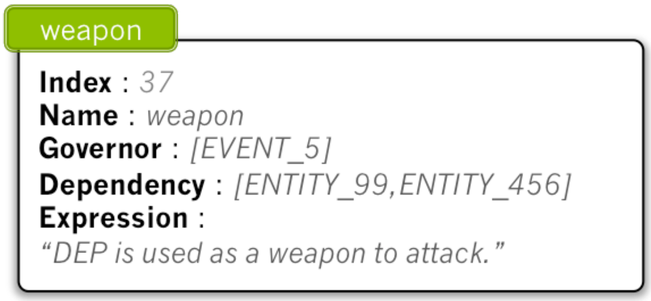
Figure 9: Relations of the Ontology

Each relation class is further defined in the ontology (example in Figure 10).

We can use a Davidsonian style representation to write knowledge with our ontology. Instances are then associated to a variable name (e for events, x for entities) and Relations are represented by predicates on these variables. For example, an entity x_1 dropping an item x_2 is represented as

$$drop(e), dropper(e, x_1), dropped(e, x_2).$$

A piece of information can be asserted by a assertion relation of our ontology, but the existence of a set of instances linked by non-assertion relation in the knowledge database also represents a piece of information. In other words, there


weapon

Index : *37*
Name : *weapon*
Governor : *[EVENT_5]*
Dependency : *[ENTITY_99,ENTITY_456]*
Expression :
"DEP is used as a weapon to attack."

Figure 10: Example of relation: Weapon (non-assertion)

are two types of facts. The first type regards the relation between two instances (entities or events), such as a comparison, a subsumption, or a causal relation (effect or condition), expressed by an assertion relation (Figure 4 or Figure 5). The second type describes a single Minecraft Instance in the game by linking it with other instances through non-assertion relations and express for example the properties of an entity or the probability of an event to occur (Figure 6).

Gold is a type of ore.
$gold(x_1), ore(x_2)$ $\_type\_of(x_1, x_2)$

Table 4: A fact (the first type) representing a subsumption between entity classes.

As an example of possible logical inference, the following piece of information is written in our knowledge database:

If a chicken dies while on fire, it will drop cooked chicken.

Then, assuming the system has the following common sense knowledge (axiom):

If an item is dropped, the player gets it.

we can deduce the following:

If a chicken is killed by fire, the player gets cooked chicken.

A system equipped with logical inference ability can thus answer a question such as:

If a chicken dies while on fire, it drops cooked chicken instead of raw chicken.
$chicken(x_1), kill(e_1), fire(x_2), drop(e_2),$ $cooked\_chicken(x_3), raw\_chicken(x_4),$ $killed(e_1, x_1), weapon(e_1, x_2),$ $dropper(e_2, x_1), dropped(e_2, x_3),$ $\_effect(e1, e2)$

Table 5: A fact (the first type) about causality between events.

Stone can be mined with a pickaxe.
$stone(x_1), mine(e_1), pickaxe(x_2)$ $mined(e_1, x_1), tool(e_1, x_2)$

Bats usually spawn in caves.
$bat(x_1), spawn(e_1), cavern(x_2),$ $spawned(e_1, x_1), in\_environment(e_1, x_2),$

Table 6: Facts (the second type) on properties of single events.

How to obtain cooked chicken?

by the inference process described above and responds:

You should kill a chicken with fire.

3.3.1 Can our meaning representation express enough information?

A question will be answerable if the answer is present in the knowledge database (the question is theoretically answerable), and if all the pieces of information that are needed to answer can be represented with our meaning representation. We expect the contents extracted from the 3 complete websites to have a high coverage, so most of the relevant questions are theoretically answerable. The quality of our QA system will then directly depend on the quantity of relevant pieces of information (that can be used to answer players' questions) that can be represented with the meaning representation that we defined.

We manually answered 10 questions of our training corpus by locating all the related pieces of information (35 different pieces of information in the database were relevant to answer the questions), and by evaluating the difficulty to answer them using our meaning representation. The preliminary analysis of the results allows us to draw some conclusions.
Firstly, all the pieces of information that are related to a question are not necessary to construct a satisfying answer. In our annotations, only 40% of the pieces of information were necessary. The main reason for that is that most of these pieces of information bring details that are not compulsory to construct a relevant answer.
Secondly, in its entirety, the knowledge database is not highly redundant (we have chosen websites with complementary information). In our annotations, more than 60% of the pieces of information were written only once. However, if we only consider the pieces of information that were absolutely necessary to construct relevant answers, about 70% of them were redundant and sometimes appeared in both text and tables (already structured data), what should be an advantage in the structuring process of the knowledge database. Nonetheless, we will have to care about the recall of the translation process not to loose essential information.
Thirdly, for 30% of the annotated questions, information contained in natural sentences and information contained in tables had to be combined in order to construct a relevant answer. So the final QA system will have to be able to combine different types of information together. This may be part of future works, as our study focuses on structuring textual information through semantic parsing.
Fourthly, the meaning representation can only represent the information in a single sentence, and even by solving the co-reference problem with the Stanford core NLP tool, we sometimes loose or misunderstand important information by removing the context of the sentence. In particular there is a risk to generalize some facts that are only true in a specific context that is not specified in the sentence. As the pages of the websites we used for the knowledge database are

divided into sections, we believe that this problem can be at least partially solved by using the names of these sections to solve some further co-references and lacks of context. This problem may also be part of future works.
Eventually, 30% of the annotated questions were not answerable with the knowledge database alone because some pieces of information, necessary for the inference process that lead to an answer, were not written in the database. These pieces of information are axioms that are obvious for human readers but that have to be taught to the QA system. For example, the question

How do I obtain an Enchantment Table?

can not be answered with the knowledge database, unless the QA system is told that:

When the player craft an object, the object is obtained by the player.

Indeed, if the crafting recipe of the Enchantment Table is written in the database, it is not explicitly written that the player will obtain this object by following this recipe. These axioms will probably have to be taught manually, but fortunately, the simple logic of Minecraft should restrict a lot the number of such axioms to about 10 axioms.

3.4 Dataset

Finally, to finish with the collection of data about Minecraft, a semantic parser for Minecraft has to be trained to recognize instances and relations in sentences. So we built a training dataset by annotating instances and relations in sentences about Minecraft (sentences from the text of the knowledge corpus we extracted). For example, when annotating the sentence:

If the cow dies while on fire, steak will be dropped instead of beef.

we annotate the instances by putting the labels *cow*, *kill*, *fire*, *steak*, *drop*, *raw_beef* (which are instances of our ontology) on the groups of words that stand for these instances in the sentence. Then, for each pair of instances that are related by a relation of our ontology, we put a label corresponding to the relation, for example *killed* for the relation between *kill* and *cow*, or *effect* for the relation between *kill*

and *drop*.
We can represent these annotations with a Davidsonian style:
instance annotations:

$$cow(x_1), kill(e_1), fire(x_2), steak(x_3), drop(e_2), raw\_beef(x_4)$$

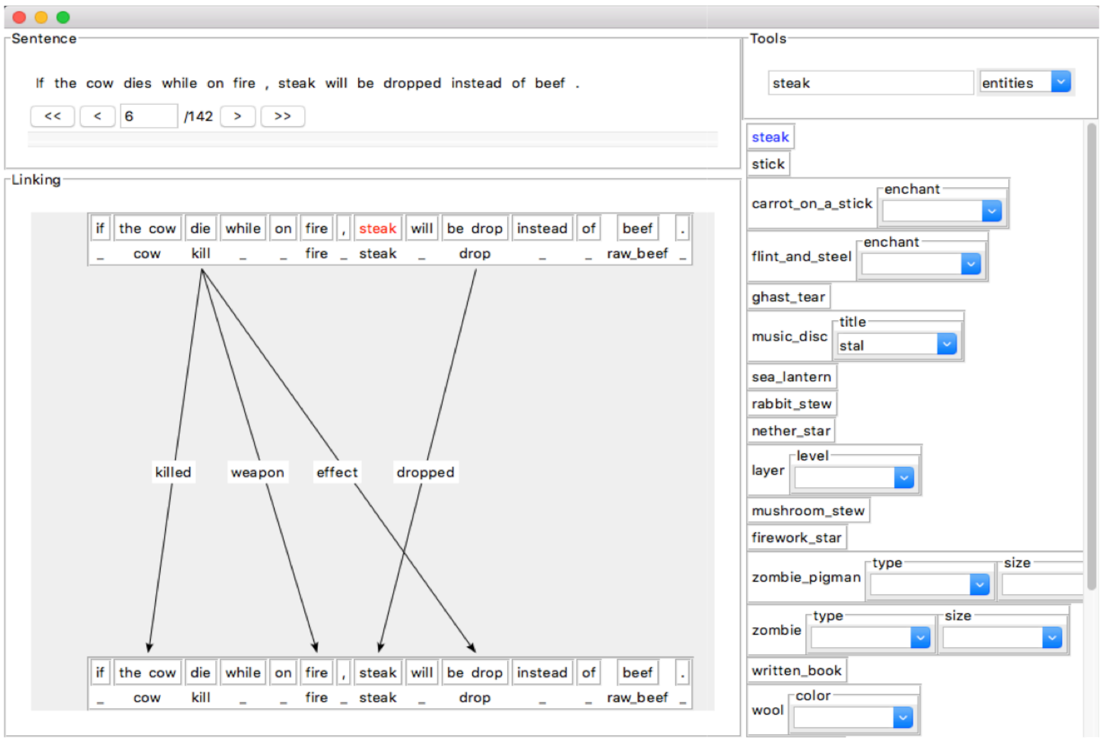
relation annotations (non-assertions):

$$killed(e_1, x_1), weapon(e_1, x_2), dropped(e_2, x_3).$$

relation annotations (assertions):

$$\_effect(e_1, e_2).$$

To do this annotation easily, we developed a specific annotation tool that integrates the ontology (see Figure 11). s it contains the ontology, this tool can


Sentence

If the cow dies while on fire , steak will be dropped instead of beef .

6 /142

Tools

steak entities

Figure 11: Annotation tool

propose the possible labels to the annotator, in order to simplify the annotation

process and to ensure that the annotation respects the ontology constraints.
Using this tool, we manually annotated 301 samples of instances (142 positive examples and 159 negative examples) and 485 samples of relations (266 positive examples and 219 negative examples).
Furthermore, we also created 6471 automatic samples of instances (positive examples only) from the anchors contained in the extracted knowledge corpus. Indeed, such as Wikipedia, the 3 websites we extracted knowledge from contain a lot of hyperlinks that lead to an other page of the same website, called anchors. These anchors then link a fragment of text to the title of an other page. If the title of the page pointed by a hyperlink can be identify as the name of an instance class in the ontology, the fragment of text that holds the hyperlink becomes a positive example for this instance class, and can be added to the training dataset.

3.5 Conclusion on the data collection

In this section, we first have described a knowledge database and a question answering corpus related to the video game Minecraft. We have shown that even for low resource domains, data can be extracted from the web, by designing adapted extracting rules.
We saw that the extracted structured data can be used to make the definition of an ontology easier.
And we saw that the textual data can be used to create a training dataset for a semantic parser, and that this training dataset can be completed automatically with instances samples by using websites anchors. We will show in the following section on semantic parsing that we can also use textual data to train embedding models that we will use to improve the performance of the semantic parser.

Our final goal is to use the collected data to build a system that can answer questions using the logic specific to the game. A lot of research has been done on the answering of real world questions using Freebase [1, 2] or Wikipedia [9]. Datasets for these tasks usually favour systems that do simple queries of facts on the knowledge database [2]. As the complexity of the questions increases, answering the questions usually becomes considerably difficult [9], due to the vast complexity of the real world. Our purpose is to show that in restricted domains with a strict logic, such as video games, even difficult questions can be addressed, and this by using only a very limited quantity of annotated resource. This will be the point of the following sections.

4 Semantic Parsing

In the context of our study, **semantic parsing** is a process that take a sentence as input and outputs a knowledge graph that represents the meaning of the sentence (see Figure 12).

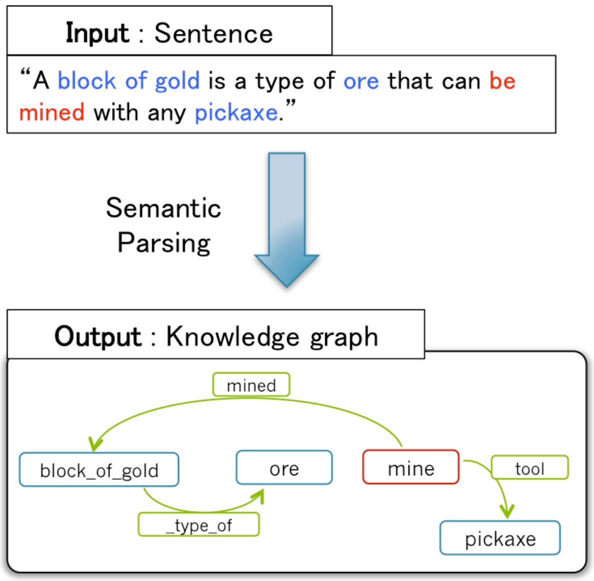


Figure 12: Semantic Parsing

The knowledge graph is composed of instances (the nodes) and relations (the edges) that belong to the ontology that we defined previously.
The knowledge graph is constructed in two steps, the instance classification that generates the nodes (in red and blue), and the relation classification that generates the edges (in green).
There is an other process called **syntactic parsing** that, like semantic parsing, generates an output graph (the syntactic tree, see Figure 13) from an input sentence, but, contrary to semantic parsing, this graph does not represent the meaning of the sentence but its syntax. Nodes are the words of the sentence, and edges are the syntactic dependencies between these words.

We can notice a similarity between the knowledge graph and the syntactic tree. Thus, our idea was to use the information contained in the syntactic tree as a base for the semantic parsing. In particular, using the words (nodes of the syntactic

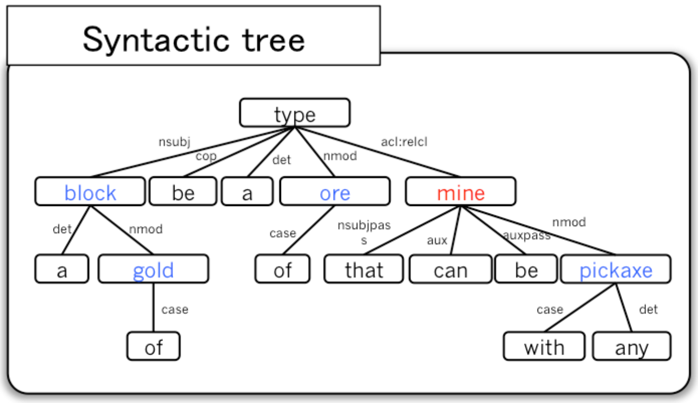


Figure 13: Syntactic tree of the sentence "A block a gold is a type of ore that can be mined with any pickaxe"

tree) in the instance classification step, and the syntactic dependencies (edges of the syntactic tree) in relation classification step.
The syntactic parsing process does not depend on a domain ontology; it has been largely studied in the past and performing syntactic parsers have been released by NLP researchers [10, 11]. So we can easily apply existing syntactic parsers to our work, and use syntactic trees in our semantic parsing.

4.1 Instance classification

Instance classification is the classification of a phrase (a group of words) into an instance class from the ontology. So we need to extract features to classify from phrases. Word embedding has been shown to produce non-sparse and effective features to represent the meaning of words [12].

4.1.1 Word Embedding

Therefore, we trained a word embedding model using as a training corpus the knowledge corpus extracted before, which is a corpus of words sequences specific to Minecraft, with the skip-gram model of the Word2Vec library [13]. We obtain a model that produce similar vectors for words with a similar meaning in the context of Minecraft.

Then, the embedding representation of a phrase using our Word2Vec model is

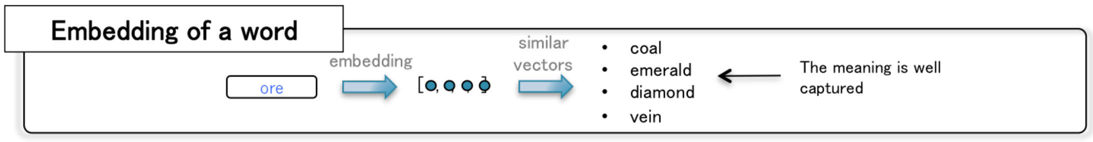


Figure 14: Embedding of the word “ore”

calculated as the average vector of the embedding representation of all words in the phrase.
The challenging point is to create, from a word embedding model, a model that embed correctly the meaning of a phrase. Our Word2Vec based phrase embedding model does not include the different importance of the words in natural language (e.g. in “a baby villager”, the word “villager” is more important to catch the meaning of the instance described than the word “baby”). This difference between the importances of each word in a phrase is visible in the syntactic tree that represents this phrase. The most important word is the root of the tree, which is specified by its children nodes, and deepest nodes have least importance. In other words, it could be interesting to include, in the embedding of a phrase, the syntactic pattern followed by the words, which is a quite difficult task [14]. However, more specifically, we can keep an information on the syntactic relations between the words by using the VecDCS library [15]. In this method, the embedding representation of a phrase is calculated using the DCS tree representation of the phrase that can be obtained from the syntactic tree (Figure 15).

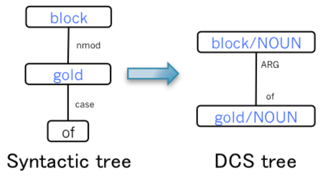


Figure 15: DCS tree of the phrase “block of gold”

And the VecDCS embedding of the phrase can be calculated from the individual embedding vectors of the different nodes of the DCS tree by the formula given in Figure 16.

Therefore, we also trained a VecDCS model for Minecraft, using the extracted knowledge corpus. We dispose of two different phrase embedding models that

$$v_{node} := v_{node} + \frac{1}{n}\sum_{child}^{n} v_{child} M_{ul_{child}} M_{ol_{child}}^{-1}$$

$v_{node},\ v_{child}$: *vectors*
$ol_{child},\ ul_{child}$: *child labels*
$M_{ol},\ M_{ul}$: *role matrices*
n : *children number*

Figure 16: Formula of the VecDCS embedding vector of a syntactic sub-tree

can be use to produce features for instance classification (Word2Vec model and VecDCS model).

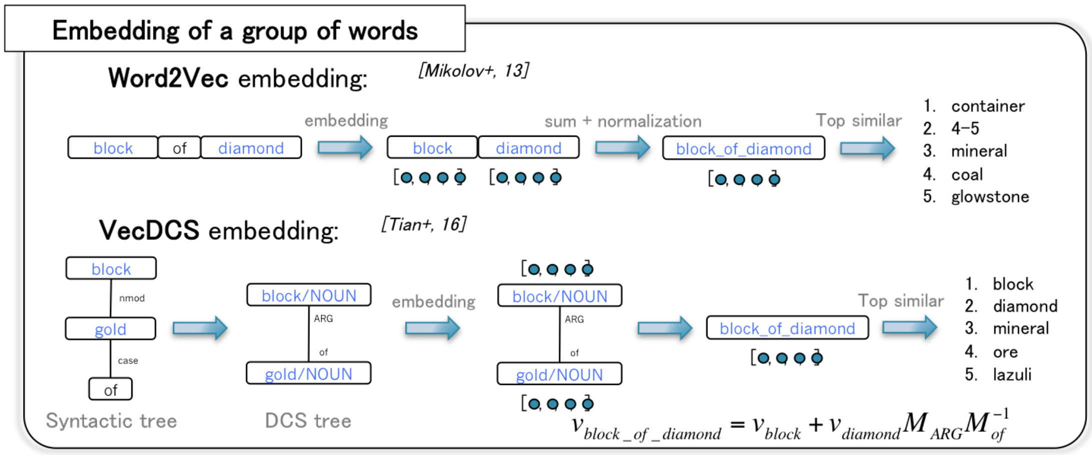


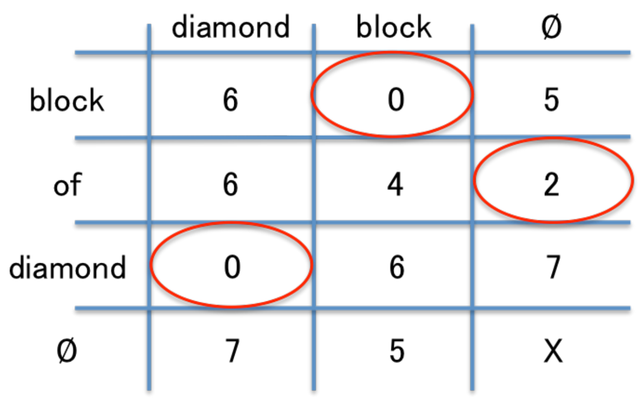
Figure 17: Two phrase embedding models

Our experiment in instance classification will, then, also be a way to study the performance of each embedding model (the VecDCS model being expected to produce more effective features).

4.1.2 Models for Instance Classification

We defined two baseline models for the instance classification step. The first one uses a string similarity method (we refer to it as StringSim). It does not use embedding models to classify words. We just compare the words of the phrase to classify with the names of every instance classes in the ontology, and chose the closest instance class as the prediction for the phrase. To calculate the string

similarity between two phrases (the name of an instance class of the ontology is also a phrase/group of words), we use a combination of the Levenshtein distance and the Hungarian algorithm (an alignment algorithm) to align words.
In details (see Figure 18), we create a matrix with as many rows as there are words in the first phrase and as many columns as there are words in the second phrase. We add an additional row and an additional column that correspond to the empty word (so that we can align a word on the empty word and have a different number of words in the two groups). Then we fill the cells of the matrix with the Levenshtein distance between words of the two phrases. Finally, we use a variation of the Hungarian algorithm to find the alignment of words that minimize the sum of the distances. The variation allows aligning any number of words (include no word) on the empty word. The result of this Hungarian algorithm on the constructed matrix is the distance between the two phrases (see Figure 18). If all words are aligned on the empty word (the cost of creating a word is smaller than the transformation of a word of the other group to this word), then there is no similarity between the two phrases and the distance in set to infinite.



	diamond	block	Ø
block	6	0	5
of	6	4	2
diamond	0	6	7
Ø	7	5	X

distance('diamond block', 'block of diamond') = 2

Figure 18: Distance between 'diamond block' and 'block of diamond'

Note that even if the distance is infinite for distant phrases, we can fix a finite upper limit on the acceptable distance between a phrase to classify and the names of classes. Over this limit, or threshold, even if a class name is the closest to the phrase among all classes, the phrase will not be classified with this class. This threshold is the only parameter of the StringSim model.

The second baseline, Flat SVM model (Figure 19), uses the simple Word2Vec embedding model to extract features from a phrase, and we classify the features with a multi-class RBF kernel SVM classifier that outputs one of the 508 instance classes of the ontology, or null if the group of words does not represent an instance.

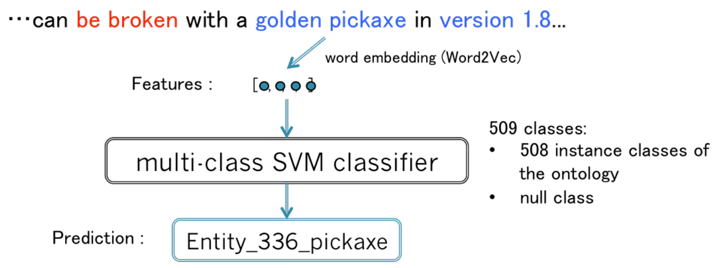

…can be broken with a golden pickaxe in version 1.8…
word embedding (Word2Vec)
Features :
multi-class SVM classifier
509 classes:
• 508 instance classes of the ontology
• null class
Prediction :
Entity_336_pickaxe

Figure 19: Flat SVM model for instance classification

This model uses a C-SVC RBF kernel SVM (support vector machine) classifier implemented in the LIBSVM library [16] (this is also the library that we used for the other models using SVM classifiers in this work). This classifier is defined by two parameters called *c* (for cost) and *gamma* (that defines the kernel function of the classifier). Hence, the training of the Flat SVM model depends on the these two hyper-parameters *c* and *gamma*.
During the training, for each class, we use positive and negative training samples. The positive samples of a class are the positive examples of the training dataset for this class and the negative samples of a class are the negative examples (not an instance) of the training dataset and the positive examples of the other classes.

Then, we defined an improved model called Hierarchical SVM model (Figure 20). This is an improvement of the Flat SVM model in which we integrate the ontology's hierarchy of instances. We still use a phrase embedding model to extract features from phrases. But we do not classify the features with one multi-class SVM classifier. We classify features recursively, following the ontology's hierarchy, using one binary RBF kernel SVM classifier for each instance class. To classify a phrase, we check at first if it belongs or not to the ENTITY class and to the EVENT class, which are the two roots of the instances hierarchy, using their respective binary classifiers. If and only if it is predicted as belonging to a class, we check the belonging to the children classes (object, version, *etc.*) and continue recursively. If the phrase doesn't belong to any children class, the recursion ends and the phrase is classified as the last predicted class. Note that we can obtain several predictions using this process. However, we can rank the several predictions by outputting the probabilities for each binary classifier's prediction. The probability of a prediction is defined as the geometric mean of the probabilities output by each binary classifier in the recursion. For example, in Figure 20, the probability of the prediction as *pickaxe* is:

$$P(Entity\_336\_pickaxe) =$$
$$(P(ENTITY) * P(object) * P(item) * P(tool) * P(pickaxe))^{1/5}$$

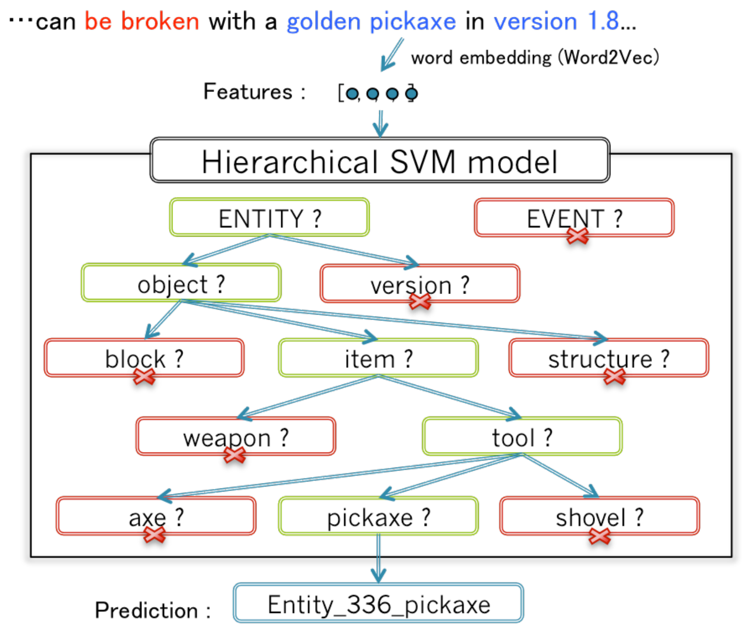


Figure 20: Improvement: Hierarchical SVM model for instance classification

As for the Flat SVM model, this model uses C-SVC RBF kernel SVM classifiers, so the training depends on two hyper-parameters *c* and *gamma*. Furthermore, as we output the probabilities of binary classifiers' predictions, we can define a threshold on the probability to decide if yes or no a phrase belong to a class. We can define a local threshold (we will refer to it as *threshold*) that must be respected by the binary classifiers' predictions at each step of the recursive

prediction, and a global threshold that must be respected by the geometric mean of all probabilities on the whole prediction (we call it cumulative threshold, and will refer to it as *cumul* for short). Then the training of the Hierarchical SVM model depends on 4 hyper-parameters: c, $gamma$, $threshold$ and $cumul$.
During the training, for each class, we use positive and negative training samples. The positive samples of a class are the positive examples of the training dataset for this class and for all the descendent classes in the ontology's hierarchy. The negative samples of a class are only the positive examples of the sibling classes and the direct parent classes, except for the roots of the hierarchy (ENITITY_0 and EVENT_0) for which the negative samples are the negative examples (not an instance) of the training dataset.

4.1.3 Tuning of Hyper-parameters

We tuned the hyper-parameters for each model by comparing their performance in terms of F1-score (micro and macro values) on the training dataset for different values of the hyper-parameters and by choosing the values that lead to the best performance. While the StringSim model's performance is measured using the whole dataset for testing, the other models have to be trained, so we measured their performance through a 2-1 cross-validation (2/3 training and 1/3 testing).

The results of the tuning for the StringSim model (Figure 21) shows that the distance threshold, when set to a small value (strict similarity), can be useful to increase the precision of the classification (ensure that the predictions are correct). However, small values of the threshold decrease the recall of the classification, because it penalizes the frequent classes that have a high lexical diversity. The values of $threshold = 1$ (strict similarity) and $threshold = 4$ (soft similarity) for the threshold have been kept for this baseline to do a comparison between the several instance classification models.

For the Flat SVM model, we tuned c and $gamma$ together by comparing the performance of the model for different couples $(c, gamma)$ (Figure 22).

We kept the values $(c = 10000, gamma = 0.1)$ to optimize the performance.

For the Hierarchical SVM model, we made the assumption that we can tune c, $gamma$ independently from $threshold$ and $cumul$ because they do not affect the

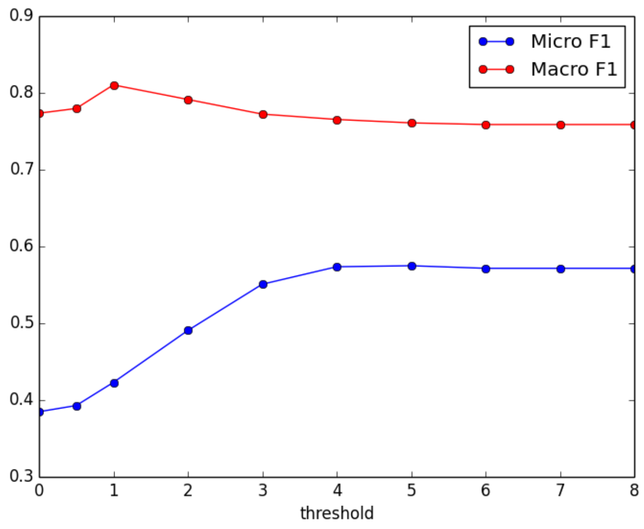


Figure 21: Distance threshold tuning for the StringSim model

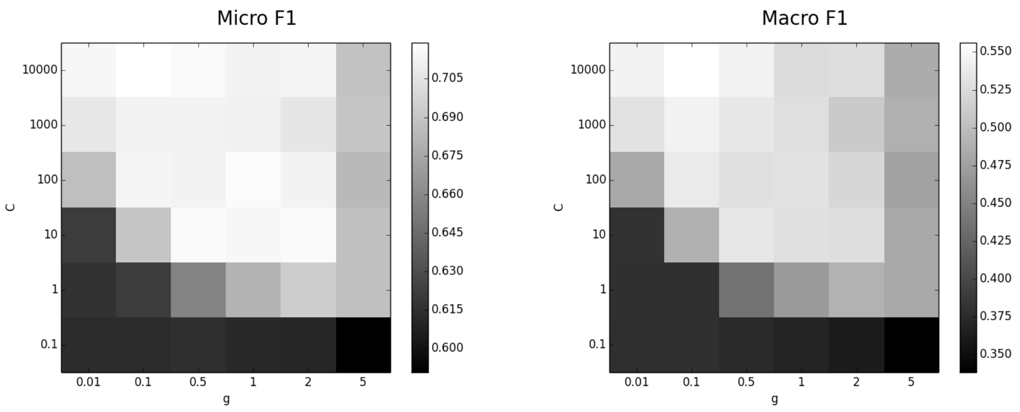


Figure 22: Hyper-parameters tuning for the Flat SVM model

same part of the model[3] and it is far easier to tune couples of hyper-parameters than tuples of 4 hyper-parameters. Then we tuned c and $gamma$ first for default values of $(threshold = 0.5, cumul = 0.5)$, and tuned $threshold$ and $cumul$ afterwards with the best values for c and $gamma$ (Figure 23).

From the results of the tuning for the Hierachical SVM model, we first chose

[3]The validity of this assumption is not evaluated in this study and should be part of future works.

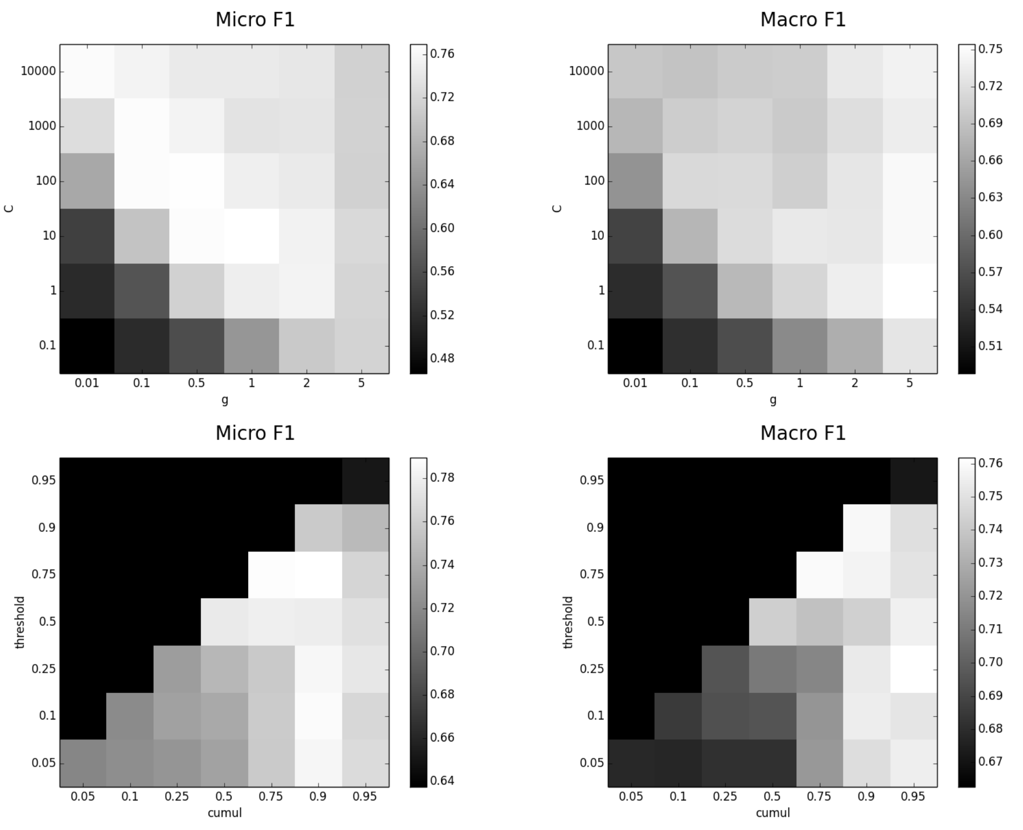


Figure 23: Hyper-parameters tuning for the Hierarchical SVM model

($c = 10, gamma = 1$) to optimize the micro F1-score (the performance on frequent classes of the ontology). We can notice that these values are not optimal in term of macro F1-score (performance giving importance to rare classes), but the loss on macro F1-score when optimizing micro F1-score is lower than the loss on the micro F1-score in the case where we optimize the macro F1-score. Furthermore, the value that is usually used for *gamma* is the inverse of the number of features used in the classification. As we use only one feature, which is the normalized embedding vector of a phrase, this value of $gamma = 1$ was expected to be optimal a priori. Then we tune the *threshold* and the *cumul* for ($c = 10, gamma = 1$). The couple ($threshold = 0.75, cumul = 0.9$) gives the best performance for micro-F1-score and one of the best macro F1-score. Note that the tuning is not done in the region $threshold > cumul$, because in this case, by definition of the cumulative threshold, any prediction that respects the local threshold at each step must necessarily respects the cumulative threshold, so *cumul* is not a relevant hyper-parameter anymore. The diagonal $threshold = cumul$ then gives the result of the tuning for the *threshold* hyper-parameter alone. On the contrary, when $threshold < cumul$, a phrase can be classified as an instance class even if the prediction from a binary classifier was low at some step of the recursion as long as the whole probability is high. Finally, we kept the values ($c = 10, gamma = 1, threshold = 0.75, cumul = 0.9$). These values will be used for the Hierarchical SVM model independently of the phrase embedding model used to produce the features to classify, which is an other strong assumption that would need to be verified in future works.

4.1.4 Experimental results for Instance Classification

We used the training dataset built before to train models (except StringSim that does not require training), and to test them through a 2-1 cross-validation. We calculated the micro F1-score and the macro F1-score for each model. Results are presented in Table 7. The micro F1-score can be considered as the performance of the model on frequent classes and the macro F1-score gives the same importance to frequent and rare classes, so the difference between micro and macro F1-scores gives an idea of the difference of performance between frequent and rare classes.

By comparing the two baselines (StringSim and Flat SVM model), we can see that the Flat SVM model, that uses embedding features is more performing on frequent classes, because frequent classes have more lexical diversity, and then classifying the meaning of words is better than classifying the surface of words. However, the Flat model uses only one SVM classifier for more than 500 classes, and as a result, rare classes are not correctly learnt even if this model has the best macro precision. This can be seen through its very low recall in Table 8, or by looking at the confusion matrix of the classification in Figure 24.

But if we compare the Flat SVM model with the Hierachical SVM model when using the Word2Vec phrase embedding model, we can see that we make a large improvement on rare classes by using the hierarchy of instances, as the gap between micro and macro F1-score largely decreases. Still, it is not sufficient to

Model	Micro F1	Macro F1
strict StringSim (threshold = 1)	0.423	0.810
soft StringSim (threshold = 4)	0.574	0.765
Flat SVM Model (Word2Vec embedding)	0.719	0.556
Hierarchical SVM Model (Word2Vec embedding)	0.791	0.761
Hierarchical SVM Model (VecDCS embedding + context features)	**0.869**	**0.855**

Table 7: Experimental results for Instance Classification

Model	Macro Precision	Macro Recall
strict StringSim (threshold = 1)	0.929	0.719
soft StringSim (threshold = 4)	0.715	**0.823**
Flat SVM Model (Word2Vec embedding)	**0.955**	0.392
Hierarchical SVM Model (Word2Vec embedding)	0.801	0.724
Hierarchical SVM Model (VecDCS embedding + context features)	0.896	0.817

Table 8: Precision and recall for Instance Classification

beat the String Similarity baseline in term of macro F1-score.
However, we could improve the performance of the Hierachical SVM model by changing the embedding model used to extract features from the phrase to classify. Instead of using the phrase embedding model based on Word2Vec word embedding alone, we use a combination of it with the more elaborate phrase embedding model based on VecDCS word embedding. The phrase embedding

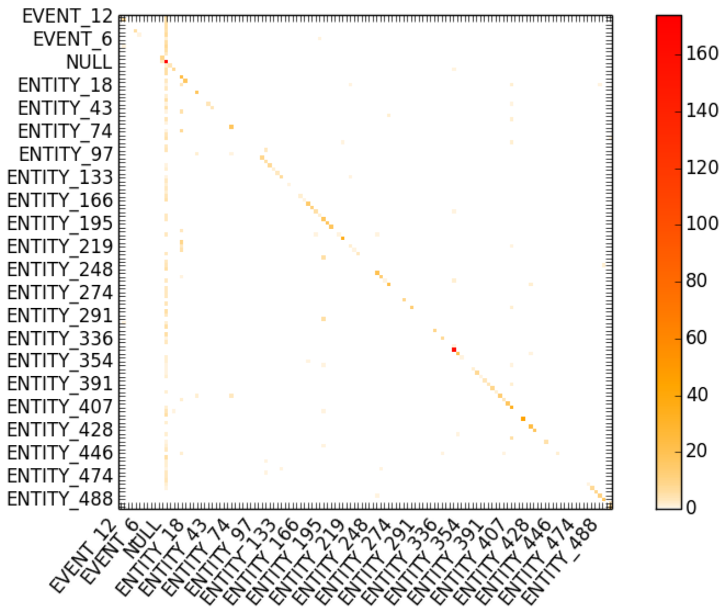


Figure 24: Confusion matrix of instance classification with the Flat SVM model

model based on VecDCS is used to extract a feature vector from the phrase to classify, and the phrase embedding model based on Word2Vec is used to extract a feature vector from the context of the phrase. To do so, we apply the Word2Vec phrase embedding model on the bag of words constituted with the words located at the nodes of the syntactic tree that are directly adjacent to at least one node that is part of the phrase to classify. The concatenation of these two vectors of features is used to classify the phrase. In this case, we use $gamma = 0.5$ as hyper-parameter, as we use two different set of features. With this model, we obtain the best performance for both micro and macro F1-scores. And if we look at the Table 8, we can see that the good performance of the Hierarchical SVM model is due to a good balance between precision and recall.

4.1.5 Conclusion on Instance classification

As a conclusion on instance classification, we have shown that:

- Word embedding models are useful in the classification of frequent instance classes with high lexical diversity.

- Using the ontology's hierarchy is useful to learn to classify correctly rare instance classes.
- We can use elaborate features (VecDCS embedding features and context features) to obtain the best performance on the instance classification task.

4.2 Relation classification

Relation classification is the classification of the syntactic paths of a syntactic tree into relation classes from the ontology. Similarly to what we saw in the section on word embedding, convert syntactic paths to their vector representation could be an efficient way to produce features to classify with a SVM model.

4.2.1 Dependency Embedding

We need an embedding model for syntactic paths that will produce similar vector representations for similar paths. The question is then, what are similar paths? As syntactic paths are sequences of words and syntactic dependencies, similar paths will be similar sequences, so it is logical to begin by embedding the elements that compose the sequences: words and syntactic dependencies. We already have a word embedding model, so we need to train a dependencyembedding model that will produce similar vectors for similar dependencies. Contrary to words, for which we can find synonyms, each dependency has a distinct syntactic role. However, some dependencies can be closer than others, such as $advcl : when$ (when condition) and $advcl : if$ (if condition), which are closer than $advcl : when$ and $advcl : to$ (consequence, or reversed condition). We can also notice the similarity between single dependencies and bigrams of dependencies (e.g. $nsubj$ and $nsubj\_xcomp^{-1}$), so it can be useful to be able to embed both unigrams and bigrams of dependencies.
To embed a dependency, or a bigram of dependencies, we used the skip-gram model implemented in the Word2Vec library (syntactic paths being composed from both words and dependencies, making our dependency embedding vectors similar to word embedding vectors simplify the implementation of the syntactic path embedding model). Training such a model only needs a corpus of dependencies sequences (in other words, syntactic paths), so we parsed a part of the

knowledge corpus we extracted before with a syntactic parser, and constituted a corpus of syntactic paths. We produced two syntactic dependency embedding models, the first one embedding dependency unigrams, the other one embedding dependency bigrams. For the second one, when constructing the training corpus of syntactic paths, we divided dependency sequences by bigrams, in order to train the embedding model on bigrams of dependencies.

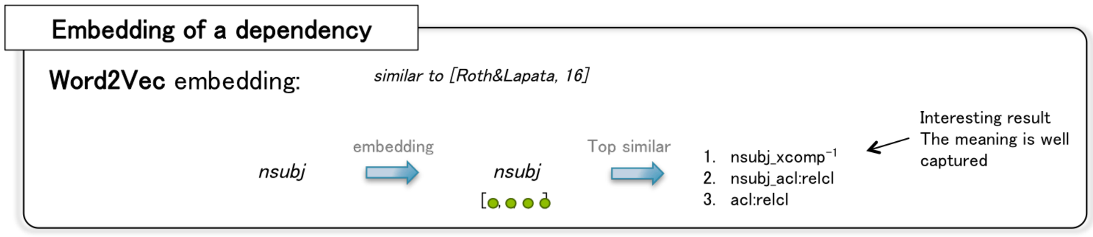


Figure 25: Embedding of a syntactic dependency unigram

To construct a syntactic paths embedding model from a word embedding model and a syntactic dependency embedding model, we begin by dividing the path into two sequences, the sequence of words and the sequence of dependencies.
We calculate the embedding vector of the sequence of words by simply calculating the average vector of the embedding representations of all the words in the sequence (the order of the sequence is not used). For the sequence of dependencies, we calculate the average vector of the embedding representations of all dependency unigrams in the path using the first embedding model (the order of the sequence is not used), or of all dependency bigrams in the path using the second embedding model (the order of the sequence is important here).
Then we concatenate the embedding representation of the sequence of dependencies and the embedding representation of the sequence of words to obtain the embedding representation of the path.

We finally dispose of two syntactic path embedding models, one that uses dependency unigram embedding and does not encode the order of the sequence, and an other that uses dependency bigram embedding and then integrates the order of the sequence of dependencies in the syntactic path. If our purpose is similar to the syntactic path embedding done in the work of M. Roth and M. Lapata in 2016 [17], the technology used to construct the model in different. It would then be interesting to compare both approaches in a future work.

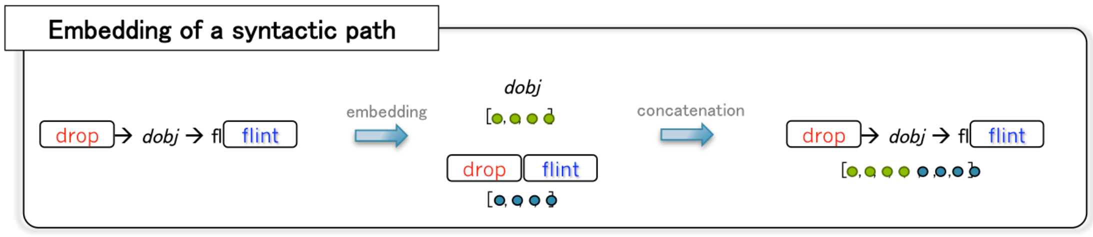


Figure 26: Embedding of a syntactic path

In our syntactic path embedding models, we also added the possibility not to consider the words in the syntactic path, in which case the path embedding models only embed the sequence of dependencies (unigrams or bigrams).

4.2.2 Models for Relation Classification

First, we created a baseline that uses one binary SVM classifier for each relation of the ontology, including the root RELATION of all relations (see Figure 27). After extracting the features from the syntactic path we want to classify with our path embedding model, we begin by testing if the path represent or not a relation of our ontology by classifying it with the root RELATION binary classifier. If the root relation's classifier prediction is positive, we test for each relation class in the ontology if the path belongs to the class or not by using the respective binary SVM classifiers (the classification is then similar to the Hierarchical SVM model for instances in the case of events that have no other hierarchy that the common root). In the example of Figure 27), only the *ingredient* relation class is predicted positively.

During the training, for each class, we use positive and negative training samples. In the case of the root relation binary classifier, we use all positive examples of the training dataset as positive samples, and all negative examples of the training dataset as negative samples. However, for any other class, the positive samples are the positive examples of the training dataset for this class and the negative samples are the positive examples of the other classes. Using a classifier for the root relation as a first step of the classification instead of testing directly all the relation classes is a way the reduce the number of predictions done by SVM classifiers, and it allows to train relation classes with less negative samples,

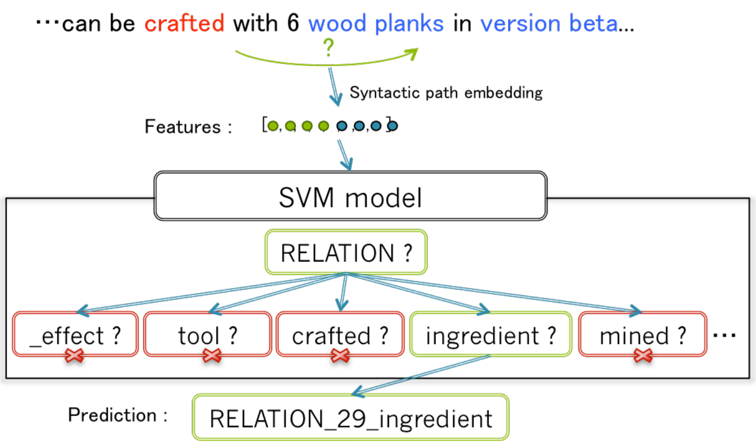


Figure 27: SVM model for relation classification

as the non-relation paths are eliminated by the root classifier at the first step.

We built an improvement of the previous model, by integrating the ontology's constraints into the model (see Figure 28). Indeed, constraints provide useful information when classifying a relation, so we added a constraints verification step at the top of the model. In other words, before classifying the features of a syntactic path between two entities, we check what relation classes are possible candidates for the relations between these two instances. In the example of Figure 28, a relation between "crafted" and "wood planks" cannot be *_effect* or *type_of*, but only *crafted* or *ingredient*. So it is theoretically useless to check if the syntactic path belong to other classes than *crafted* and *ingredient*.

As for the baseline, during the training, for any class except the root, the positive samples are the positive examples of the training dataset for this class, however the negative samples are only the positive examples of the other classes for which this class was also a candidate. The constraints verification step then allows to reduce the number of negative samples even more, which speeds up the training.
For each of the two models, we can use either the unigram-based syntactic path embedding model or the bigram-based syntactic path embedding model to produce the features to classify, and for each embedding model, we can consider or not the words of the syntactic paths. In the following section, we will compare both models, but also the difference in performance for each model depending on

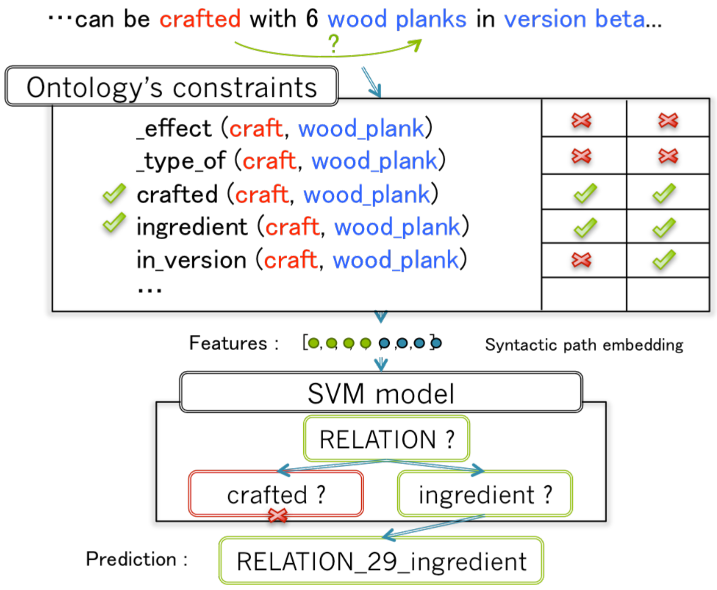

⋯can be crafted with 6 wood planks in version beta...
?
Ontology's constraints
_effect (craft, wood_plank)
_type_of (craft, wood_plank)
crafted (craft, wood_plank)
ingredient (craft, wood_plank)
in_version (craft, wood_plank)
...
Features :
Syntactic path embedding
SVM model
RELATION ?
crafted ?
ingredient ?
Prediction :
RELATION_29_ingredient

Figure 28: Improvement: SVM model with constraints for relation classification

the syntactic path embedding model that is used to extract features.
Such as the Hierarchical SVM model for instance classification, both relation classification models use C-SVC RBF kernel SVM classifiers, so the training depends on two hyper-parameters c and $gamma$. Furthermore, we output the probabilities of binary classifiers' predictions, so we can define a threshold on the probability to decide if yes or no a syntactic path belongs to a relation class. And as the prediction is done in two steps (root classifier and relation classifier), we defined a local threshold ($threshold$) that must be respected by any binary classifier, and a global threshold that must be respected by the geometric mean of the probabilities at the two steps of the prediction (we call it cumulative threshold, $cumul$). Then the training of both models depends on 4 hyper-parameters: c, $gamma$, $threshold$ and $cumul$.

4.2.3 Tuning of Hyper-parameters

As for the instance classification Hierarchy SVM model, we made the assumption that we can tune c, $gamma$ independently from $threshold$ and $cumul$. We also made the assumption that the tuning can be done only ones for both models as the difference is in the constraints verification step and not in the SVM classifiers. Then we tuned c and $gamma$ first for default values of ($threshold = 0.2, cumul = 0.2$), and tuned $threshold$ and $cumul$ afterwards with the best values found for c and $gamma$.

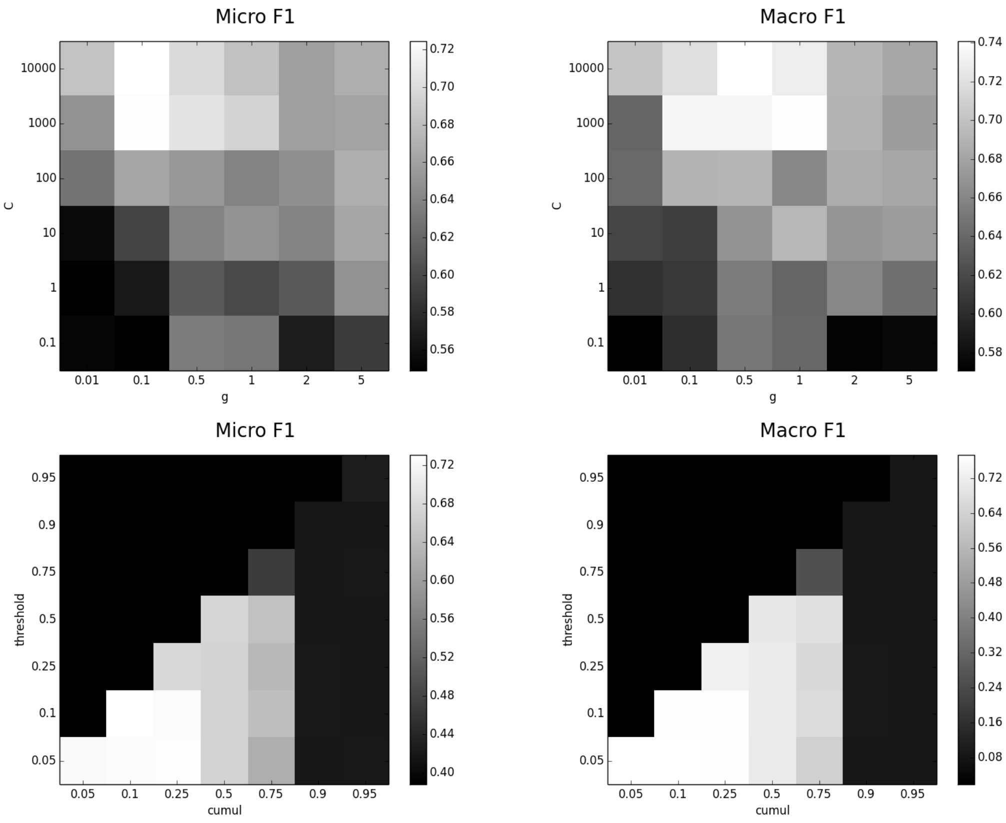


Figure 29: Hyper-parameters tuning for the Relation Classification models

From the results of the tuning, we first chose ($c = 10000, gamma = 0.5$) to optimize both micro and macro F1-score. Then we tune the $threshold$ and the $cumul$ for ($c = 10000, gamma = 0.5$). For both micro and macro F1-scores, the couple ($threshold = 0.1, cumul = 0.1$) gives the best performance. Note that $threshold = cumul$ means that $cumul$ is a useless hyper-parameter as it will always be respected. Indeed, the geometric mean of two probabilities that must be higher than $threshold$ will also be higher than $threshold$, and then higher than $cumul$. This is also the reason why the tuning is not done in the region $threshold > cumul$. Finally, we kept the values ($c = 10, gamma = 1, threshold =$

$0.1, cumul = 0.1)$.

4.2.4 Experimental results for Relation Classification

We used the training dataset built before to train and to test our relation classification models (through 2-1 cross-validation). The result of this experiment in presented in Table9.

Model	Micro F1	Macro F1
SVM Model no constraints, dependency unigrams	0.496	0.447
SVM Model no constraints, dependency bigrams	0.501	0.456
SVM Model with constraints, dependency unigrams	0.767	0.789
SVM Model with constraints, dependency bigrams	**0.792**	**0.791**
SVM Model with constraints, words and dependency bigrams	0.702	0.638

Table 9: Experimental results for Relation Classification

This experiment confirms the positive impact of using constraints of the ontology in the relation classification model. Indeed, using constraints reduces the number of possible labels, so it also reduces the number of possible classification errors. The improvement is largely visible by comparing the confusion matrices of the models without and with constraints (Figure 30)
An other interesting result is that using a constraints verification step in the model implies that we do not need to train the binary SVM classifiers by using the samples of other relation classes as negative samples if the two relations do not link the same instance classes. This results in a faster training, as we reduce the number of negative samples that are necessary.
We can also observe that using the bigram-based syntactic path embedding model to extract the features slightly increases performance, which shows that the order

Model	Macro Precision	Macro Recall
SVM Model no constraints, dependency unigrams	0.444	0.450
SVM Model no constraints, dependency bigrams	0.474	0.440
SVM Model with constraints, dependency unigrams	**0.884**	0.713
SVM Model with constraints, dependency bigrams	0.871	**0.725**
SVM Model with constraints, word and dependency bigrams	0.862	0.506

Table 10: Precision and recall for Relation Classification

in the sequence of dependencies is important.
However, using both the words and the syntactic dependencies of the path to classify in the features is less performing than using the dependencies only. This can be explained by two factors. First, we are considering here the constrained SVM model in which the number of candidates in the classification in very limited. Such a small number of possible candidates may explain that the dependencies alone are sufficient to make the distinction between these few candidates, and it may be the reason why performance does not increase when using the words of the path. Secondly, we are working with very low resource, and our training set is small. Then, there may be enough samples to learn the typical syntax used by each relation, but not enough to learn the diversity of words that can be used with this syntax. A further study with a larger training dataset might show some improvements in using words in the syntactic paths classification, but as our purpose is to work with low resource, this result encourages to develop models that use the information of words more efficiently, without a need to increase the size of the training dataset.
Looking at the Table10, we can see that the main weakness of our relation classification models is the low recall of the process. As we are working with low

resource and few training samples, making improvements on the recall without increasing the number of costly expert annotations would be valuable. We will see in the next section how we can increase the size of the training dataset without doing expert annotations of relations by using crowd-sourcing.

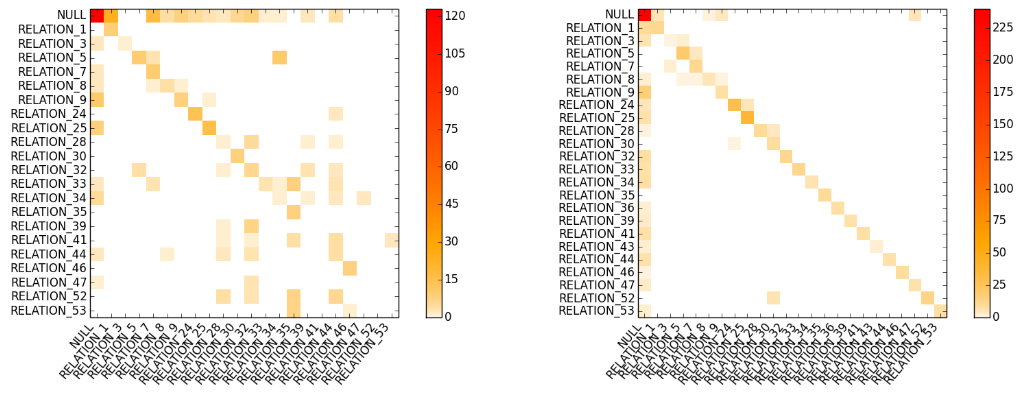


Figure 30: Confusion matrices of relation classification using dependency bigrams features without constraints (left) and with constraints (right)

We made a second experiment to show the impact of the quality of the syntactic information that is classified. We saw that our relation classification models classify syntactic paths that are obtained from sentences with syntactic parsers. In this experiment, we use the same relation classification model (SVM model with constraints, using either dependencies unigrams or bigrams, but not words) and the same training dataset, but we use two different syntactic parsers to obtain the syntactic paths from which we extract the features to classify (see Table 11).

The two parsers we use are the Stanford Parser [10] and the Parsey McParseface Parser [11]; the last one have been shown to achieve state-of-the-art performance on the syntactic parsing task.

As a result, we observe a significant difference in the performance of the relation classification model, with better performance when using the Parsey McParseface parser to obtain the syntactic paths. This is mainly due to the fact that even a small difference in the syntactic tree of a sentence can lead to big changes in the syntactic paths and make them more difficult to classify correctly with our relation classification models. Therefore, using a syntactic parser that avoid

Model	Micro F1	Macro F1
SVM Model with constraints, dependency unigrams (*Parsey McParseface* Parser)	0.767	0.789
SVM Model with constraints, dependency bigrams (*Parsey McParseface* Parser)	**0.792**	**0.791**
SVM Model with constraints, dependency unigrams (*Stanford* Parser)	0.731	0.774
SVM Model with constraints, dependency bigrams (*Stanford* Parser)	0.774	0.784

Table 11: Impact of the syntactic information quality on Relation Classification performance

mistakes has a significant positive impact on the relation classification model's performance.

4.2.5 Conclusion on Relation classification

As a conclusion on relation classification, we have shown that:

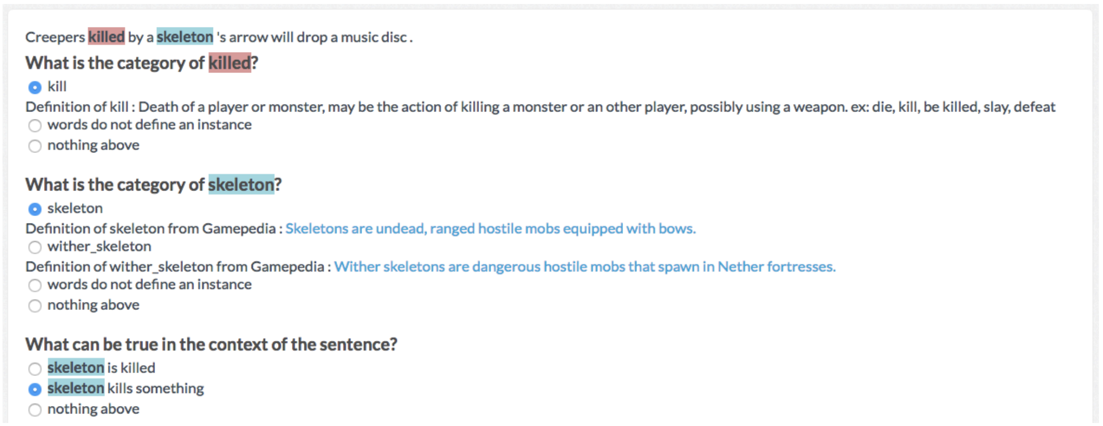
- A dependency embedding model can be trained from the extracted corpus (after parsing it with a syntactic parser) and allows to classify syntactic paths.
- Using the ontology's constraints increases performance and speed up the training.
- Using dependency bigrams instead of unigrams, and then taking into account the order of the dependencies in the syntactic path slightly increases performance.
- Using both words and dependencies from the paths to classify decreases

performance, which shows that we must re-think the way we are using the word information in the path classification.

- The quality of the syntactic information that we classify in the Relation classification step has a significant impact on performance.

4.3 Crowd-Sourcing

We saw in the previous sections that working with low resource and small training datasets implies a low recall in the classifications. If we have been able to generate automatically a lot of instances samples to train the instance classification models by using the anchors of the websites we used to extract knowledge about Minecraft, we do not dispose of such a way to generate easily and cheaply training samples for the relation classification yet. This results in a quite low recall in the relation classification step of the semantic parsing. One solution is to use crowd-sourcing to obtain cheap annotations made by non-expert annotators on the web to increase the size of the training dataset for relation classification. However, annotating relations between two instances in a text is a difficult task that requires knowing the characteristics of each relation class in the ontology. To make non-expert annotators to realize such an annotation, we need to simplify the annotation task. What we did is that for each relation sample to be annotated, we generated automatically a short list of propositions in natural language (English) that are easy to understand for human annotators. When annotating the relation between two already automatically classified instances, we begin by building the list of all possible relation class that can be used to link these two instances together. These lists are always rather short, between 1 and about 5 candidates, by construction of the ontology. Then, after having shown the sentence to the annotator, we show him, for each possible relation class, an English proposition that asserts that the relation is true in the context of the sentence and ask him to choose the only proposition that is true (the ontology is designed in a way that there is always at most only one relation between two instances). In the case where there is no relation between the couple of instances (this is a negative sample), the annotator has the possibility to answer that none of the propositions is true. We also ask to the annotator to confirm that the instances have been correctly classified before displaying the propositions in order to avoid wrong annotations of the relation, as wrong instances may lead to a wrong list of possible relation candidates. Each sample is annotated by 5 different annotators, and the most reliable answer is added to the training dataset. We obtained 210 distinct relation samples (158 positive examples, and 52 negative examples) for 9 different relation classes. This crowd-sourcing has been realized with the CrowdFlower platform[4] (example in Figure 31).


Creepers killed by a skeleton 's arrow will drop a music disc .

What is the category of killed?

- [✓ kill]

Definition of kill : Death of a player or monster, may be the action of killing a monster or an other player, possibly using a weapon. ex: die, kill, be killed, slay, defeat

- words do not define an instance
- nothing above

What is the category of skeleton?

- [✓ skeleton]

Definition of skeleton from Gamepedia : Skeletons are undead, ranged hostile mobs equipped with bows.

- wither_skeleton

Definition of wither_skeleton from Gamepedia : Wither skeletons are dangerous hostile mobs that spawn in Nether fortresses.

- words do not define an instance
- nothing above

What can be true in the context of the sentence?

- skeleton is killed
- [✓ skeleton kills something]
- nothing above

Figure 31: Example of crowd-sourcing annotation with CrowdFlower

We evaluated the quality of the annotated data by comparing the performance of the relation classification cross-validation on the crowd-sourcing annotations to the performance of cross-validation on expert annotations (see Table 12).

If the performance of the cross-validation on crowd-sourcing annotations is slightly below the performance on expert annotations, we can nonetheless conclude that crowd-sourcing can be a good way of increasing the size of the training dataset for relation classification when using the annotating method described before to make the annotation accessible to non-expert annotators.

[4] www.crowdflower.com

Model	Micro F1	Macro F1
SVM Model with constraints, dependency bigrams (expert annotations only)	**0.792**	**0.791**
SVM Model with constraints, dependency bigrams (crowd-sourcing only)	0.763	0.719
SVM Model with constraints, dependency bigrams (expert + crowd-sourcing)	0.780	0.754

Table 12: Relation Classification performance on crowd-sourcing annotations

5 Question Answering

We dispose of a semantic parser specific to the domain of Minecraft. This semantic parser output knowledge graphs from English sentences, so an interesting issue would now be to study how we can use knowledge graphs to do question answering in Minecraft, and what kind of logical reasoning can be done on knowledge graphs to retrieve answers to difficult questions.
This section has for only goal to expose some ideas about a method that can be used to do QA in Minecraft by using the knowledge graphs, and it does not provide any proof or result. Such results could be obtained in a future study, by applying the proposed QA algorithm to the question corpus that have been extracted.

5.1 Knowledge graphs and question graphs

We begin with the observation that the semantic parser we built to convert English sentences about Minecraft into knowledge graphs can be as well on the questions. In the short study of this section on question answering, we will divide questions into two categories: question-words questions and yes/no questions that do not contain question words.
When applying the semantic parser to a yes/no question (Figure 32), we can see

that the question graph is almost identical to the knowledge graph of the corresponding assertion. We will then be able to answer yes/no questions by looking for knowledge graphs similar to the question graphs in the knowledge database. The presence of such a similar knowledge graph giving the answer “Yes”, and the absence giving the answer “No”.

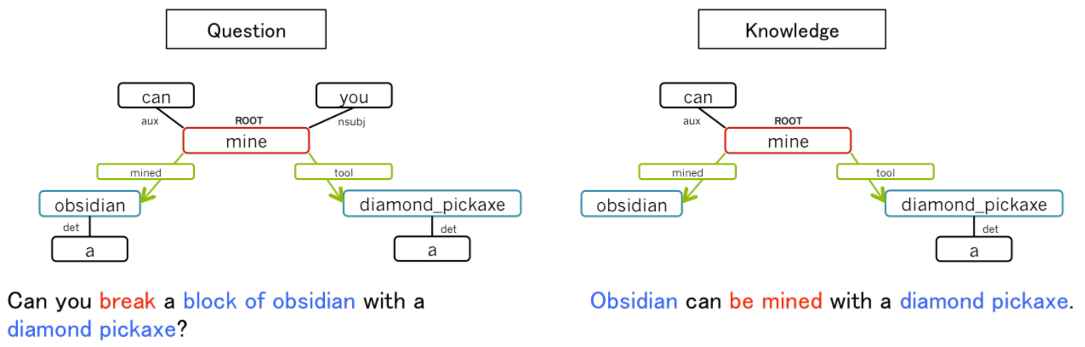


Figure 32: Example of yes/no question graph and the corresponding assertion graph

When applying the semantic parser to a question-word question (Figure 33), we can see that the question graph is similar the knowledge graph containing the answer, with the difference that the question graph has an **unknown instance node** in the place of the sub-graph that represents the answer. We will then be able to answer question-words questions by looking in the knowledge database for compatible sub-graphs that can be substituted to the unknown instance node.

5.2 QA algorithm

Several studies have been done on using knowledge graphs to do question answering. Some of them developed methods to retrieve answers from graphs without the need of an ontology, and can be done in open-domain [18, 19] because they do not require to convert both question and knowledge to semantic graphs. These algorithms provide an interesting alternative to our method.
However, in the case of our study, we are more interesting in the works that make use of both question graphs and knowledge graphs, and developed methods

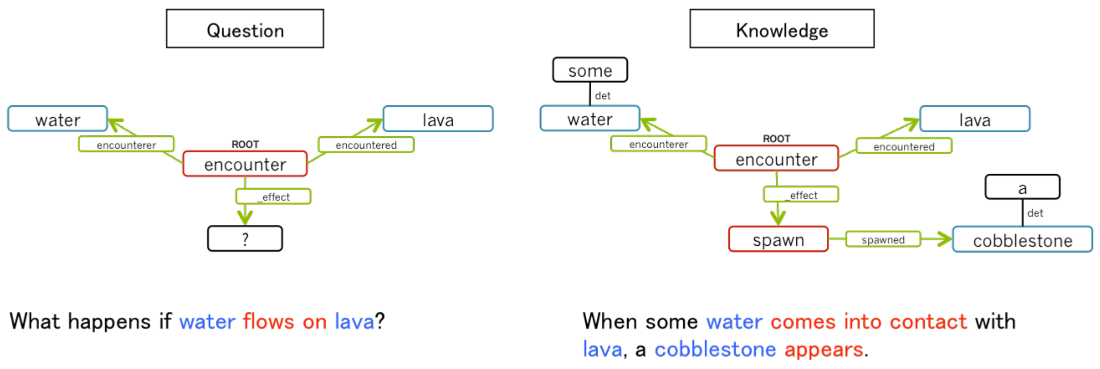


Figure 33: Example of question-word question graph and the knowledge graph of its answer

to retrieve answers by graph comparison (common paths and overlaps) between the questions graphs and the knowledge graphs containing candidate answers [20, 21, 22]. The algorithm we describe in this section (Algorithm 1,2) is closely related to these last works.

For yes/no questions, answering a question consists in comparing the question graph with every knowledge graph in the knowledge database. If any knowledge graph overlaps with the question graph more than a fixed threshold, the answer is “yes”, otherwise it is “no”.

In the case of question-words questions, retrieving answers is done in two steps. In the first step we retrieve candidates answers. To do this, we consider the path between the root and the unknown instance node in the question graph. Any knowledge graph that contains this path provides a candidate answer, which is the sub-graph at the end of this path. Then, in the second step, for each candidate answer, we replace the unknown instance node by the candidate answer sub-graph, and we compare the modified question graph with the knowledge graph the candidate answer is extracted from. If the overlaps exceeds a fixed threshold, the candidate answer is validated.

Note that the obtained answers are semantic graphs, so it is possible to do logical reasoning, by combining different pieces of knowledge from different knowledge graphs to answer. To do this, after having retrieved the answer, we convert it into a new question graph by adding an unknown instance to it and we repeat the algorithm to retrieve answers. An example in given the Appendix A.

Algorithm 1 QA algorithm - step 1

```
procedure RetrieveCandidates($questionGraph$,$knowledgeGraphCollection$)
    $candidates \leftarrow []$
    if $questionGraph$ is $yesNoQuestion$ then
        for all $knowledgeGraph$ in $knowledgeGraphCollection$ do
            if $questionGraph.root$ in $knowledgeGraph.nodes$ then
                $candidate[0] \leftarrow$ "Yes"
                $candidate[1] \leftarrow questionGraph$
                $candidate[2] \leftarrow knowledgeGraph$
                $candidates$.append($candidate$)
    else if $questionGraph$ is $qWordQuestion$ then
        $rootIndex \leftarrow questionGraph.root$
        $qWordIndex \leftarrow questionGraph$.index($qWord$)
        $path \leftarrow questionGraph$.path($rootIndex$, $qWordIndex$)
        for all $knowledgeGraph$ in $knowledgeGraphCollection$ do
            if $path$ in $knowledgeGraph.paths$ then
                $candidateNode \leftarrow knowledgeGraph$.paths($path$)$.end$
                $answer \leftarrow candidateNode.subgraph$
                $candidate[0] \leftarrow answer$
                $candidate[1] \leftarrow questionGraph$.replace($qWordIndex$,$answer$)
                $candidate[2] \leftarrow knowledgeGraph$
                $candidates$.append($candidate$)
    return $candidates$
```

Algorithm 2 QA algorithm - step 2

```
procedure ValidateAnswers($candidates$)
    for all $candidate$ in $candidates$ do
        $answer \leftarrow candidate[0]$
        $modifiedQuestionGraph \leftarrow candidate[1]$
        $originKnowledgeGraph \leftarrow candidate[2]$
        if overlap($modifiedQuestionGraph$,$originKnowledgeGraph$)$>$ $threshold$ then
            Print "Answer: ", $answer.text$
            Print "Justification: ", $originKnowledgeGraph.text$
```

6 Conclusion

In this work, we presented a study about semantic parsing with very low annotated resource in the domain of the video game Minecraft. We described the methods used to collect data about the domain and the classification models that we designed to build a semantic parser. We have shown that such a semantic parser can be performing in the restricted domain if we make good use of the data we created, in particular by using the ontology's characteristics, and that it can be trained with only a few manually annotated resource and can be completed with automatically generated samples or through crowd-sourcing annotations. We also described a question answering algorithm that allows logical reasoning on the knowledge graphs output by the semantic parser to answer non-factoid questions.
Our study brings two main contributions when compared to other works describing methods to build semantic parsers for limited domains [23]. First, we explored various ways to increase the parsing performance by acting on both the size of the training dataset with several methods (web anchors, crowd-sourcing), and the classification models (integrating ontology in the classification step and extracting relevant features). And secondly, our semantic parser output semantic graphs that are immediately usable by a QA algorithm to answer even difficult questions using logical reasoning.
Thus, we believe that our work is then a first step in answering to how to efficiently build a semantic parser with very low resource and use it to do advanced question answering in restricted domains.

As future works, we believe that some work should be done to improve the relation classification step of the semantic parsing. Indeed, the recall still has to be reduced, which implies a lot of manual annotations with the current methodology. One interesting problematic could be to answer the question: can we create training data automatically from available resource, as we did with anchors for instance classification? We could for example imagine that we look for typical couples of instances that can only be linked by one possible relation in a natural language corpus to create automatically relations samples.
We think that some work should also be done to reduce even more the quantity

of manual work needed to build the ontology of the restricted domain. There have been some interesting works on how a system can learn to produce knowledge graphs without any ontology [19] that have shown that the performance are particularly good in restricted domains. Or may even be possible to limit semantic parsing to instance classification and do question answering directly on syntactic trees [22]. Then, a deeper study should be done on the application of the knowledge graphs produced by semantic parsing on question answering in Minecraft. The algorithm should be improved by inspiring from other studies on graph-based question answering systems [20, 21] and tested on the corpus of questions that has already been collected.
Finally, it would be interesting to use our models to train and test a semantic parser and a QA system on other low resource restricted domains [24] such as technical fields (medical domain, industrial domains, *etc.*), customer services, or other video games.

Acknowledgements

I would first like to express my gratitude to my supervisor Professor Kentaro Inui from the Graduate School of Information Sciences at Tohoku University. Professor Inui has been of great help, not only for supervising my research, but also for my student life in general. My master thesis in Tohoku University would not have been possible without him.
I am also deeply indebted to Research Assistant Professor Ran Tian and Associate Professor Naoaki Okazaki for their continuous guidance over the duration of my research. I would like to thank them for their implication, and for the time they spent giving me the advices and inspiration that oriented my work.
I would like to thank Professor Yoshifumi Kitamura and Professor Tetsuo Kinoshita who were involved in the validation of this research project.
Furthermore, I would like to thank the people from the Incoming Student Exchange Section and the JASSO scholarship program very deeply for their kindness and their involvment. The JASSO scholarship really allowed me to integrate better in the Japanese society, as I could participate to more events and sometimes travel with friends to discover new beautiful places in Japan.
Finally, I would also like to thank all the Inui-Okazaki laboratory members, professors, researchers and students, for their valuable comments and support, and for having created together a very pleasant working environment.

References

[1] Jonathan Berant and Percy Liang. Semantic parsing via paraphrasing. In *ACL (1)*, pages 1415–1425, 2014.

[2] Xuchen Yao. Lean question answering over freebase from scratch. In *HLT-NAACL*, pages 66–70, 2015.

[3] SRK Branavan, Nate Kushman, Tao Lei, and Regina Barzilay. Learning high-level planning from text. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, pages 126–135. Association for Computational Linguistics, 2012.

[4] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *International joint conference on artificial intelligence (IJCAI)*, page 4246, 2016.

[5] Akira Fujita, Akihiro Kameda, Ai Kawazoe, and Yusuke Miyao. Overview of todai robot project and evaluation framework of its nlp-based problem solving. *World History*, 36:36, 2014.

[6] Nate Kushman, Yoav Artzi, Luke Zettlemoyer, and Regina Barzilay. Learning to automatically solve algebra word problems. Association for Computational Linguistics, 2014.

[7] Dipendra Kumar Misra, Kejia Tao, Percy Liang, and Ashutosh Saxena. Environment-driven lexicon induction for high-level instructions. In *ACL (1)*, pages 992–1002, 2015.

[8] Ran Tian, Yusuke Miyao, and Takuya Matsuzaki. Logical inference on dependency-based compositional semantics. In *ACL (1)*, pages 79–89, 2014.

[9] Panupong Pasupat and Percy Liang. Compositional semantic parsing on semi-structured tables. *arXiv preprint arXiv:1508.00305*, 2015.

[10] Danqi Chen and Christopher D Manning. A fast and accurate dependency parser using neural networks. In *EMNLP*, pages 740–750, 2014.

[11] Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally normalized transition-based neural networks. *arXiv preprint arXiv:1603.06042*, 2016.

[12] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[14] Sho Takase, Naoaki Okazaki, and Kentaro Inui. Modeling semantic compositionality of relational patterns. *Engineering Applications of Artificial Intelligence*, 50:256–264, 2016.

[15] Ran Tian, Naoaki Okazaki, and Kentaro Inui. Learning semantically and additively compositional distributional representations. *arXiv preprint arXiv:1606.02461*, 2016.

[16] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[17] Michael Roth and Mirella Lapata. Neural semantic role labeling with dependency path embeddings. *arXiv preprint arXiv:1605.07515*, 2016.

[18] Ahmad Aghaebrahimian and Filip Jurcıcek. Open-domain factoid question answering via knowledge graph search.

[19] Ben Hixon, Peter Clark, and Hannaneh Hajishirzi. Learning knowledge graphs for question answering through conversational dialog. In *HLT-NAACL*, pages 851–861, 2015.

[20] Diego Mollá and Menno Van Zaanen. Learning of graph rules for question answering. In *Proceedings of the Australasian Language Technology Workshop*, volume 92, 2005.

[21] Diego Mollá. Learning of graph-based question answering rules. In *Proceedings of the First Workshop on Graph Based Methods for Natural Language Processing*, pages 37–44. Association for Computational Linguistics, 2006.

[22] Helena Gómez-Adorno, Grigori Sidorov, David Pinto, and Alexander F Gelbukh. Graph based approach for the question answering task based on entrance exams. In *CLEF (Working Notes)*, pages 1395–1403. Citeseer, 2014.

[23] Yushi Wang, Jonathan Berant, Percy Liang, et al. Building a semantic parser overnight. In *ACL (1)*, pages 1332–1342, 2015.

[24] Diego Mollá and José Luis Vicedo. Question answering in restricted domains: An overview. *Computational Linguistics*, 33(1):41–61, 2007.

Appendix

A Question Answering: example of logical reasoning on knowledge graphs

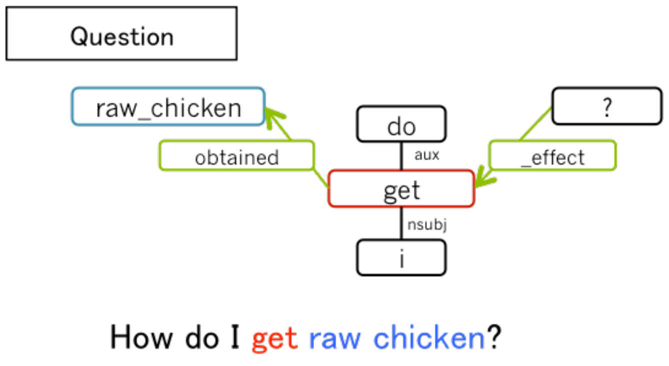


Figure 34: Question graph

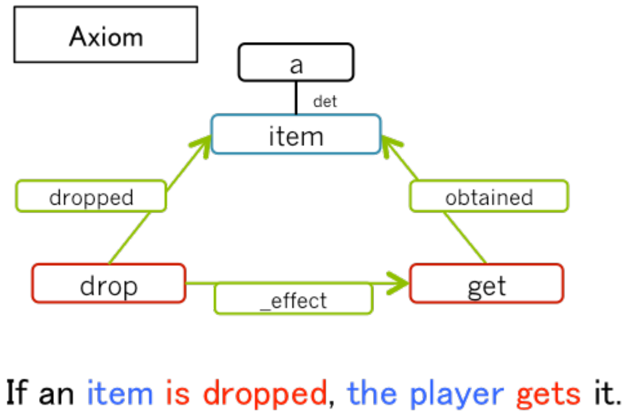


Figure 35: Axiom

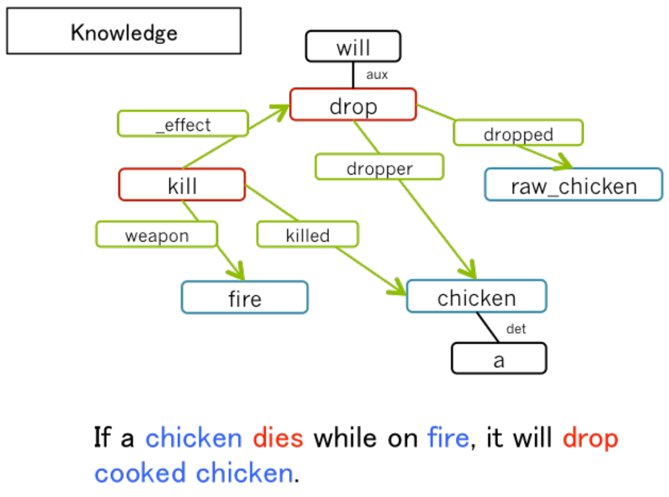


Figure 36: Piece of knowledge

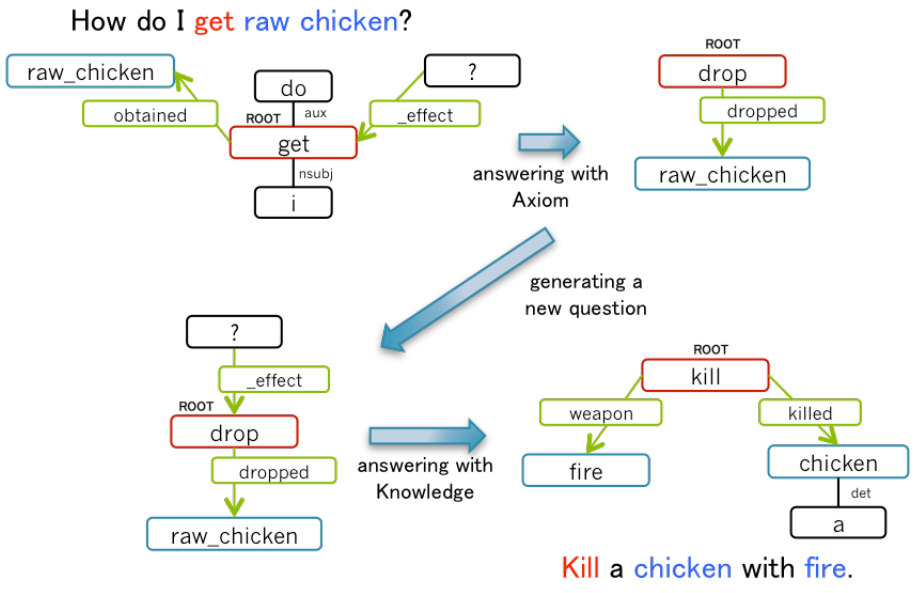


Figure 37: Logical reasoning to retrieve the answer

List of Publications

International Conferences Papers

- Corentin Dumont, Ran Tian, and Kentaro Inui. question answering with logic specific to video games. Language Resources and Evaluation Conference, 2016.

Other Publications

- Corentin Dumont, Ran Tian, and Kentaro Inui. An ontology for question-answering on minecraft. The Association for Natural Language Processing, 2016.