Understanding English with Lattice-Learning

by

Michael Tully Klein, Jr.

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May, 2008

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2008.

Certified by .....
Patrick Winston, Ford Professor of Artificial Intelligence and Computer Science
M.I.T. Thesis Supervisor

Accepted by .....
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Understanding English with Lattice-Learning

by

Michael Tully Klein, Jr.

Submitted to the Department of Electrical Engineering and Computer Science

May 23, 2008

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

A computer program that can understand the meaning of written English must be tremendously complex. It would break the spirit of any programmer to try to code such a program by hand; the range of meaning we can express in natural language is far too broad, too nuanced, too filled with exception. So I present UNDERSTAND, a program you can teach by example. Learning by example is an engineering expedient: it is much easier for us to come up with specific examples of a concept than some sort of perfect Platonic model. UNDERSTAND uses a technique I call Lattice-Learning to generalize accurately from just a few examples: "Robins, bees and helicopters can fly, but cats, worms and boats cannot," is enough for UNDERSTAND to narrow in on our concept of flying things: birds, insects and aircraft. It takes only 8 positive and 4 negative examples to teach UNDERSTAND how to interpret sentences as complicated as "The cat ran from the yard because a dog appeared." UNDERSTAND is implemented in 2300 lines of Java.

Acknowledgements

Mom, Dad, Lisa, and Jen, for expecting only that I make myself proud. Without you I would not be me! Some people write so their peers can understand; some people write so everyone can understand; I only care that you understand.

Patrick Winston, for teaching me that the best ideas are simple ideas, and for insisting that those simple ideas are worth exactly how much you help other people see them.

The Genesis Group, especially Adam Kraft, Mark Seifter, and Sam Glidden, for reflecting much, much smarter ideas back at me than I ever give to you.

Richard Feynman, Howard Roark, and Gaius Baltar, for being different sorts of maniacs, all that I aspire to become.

The Manhattan Project, for teaching me to take all my notes in pen.

Contents

List of Figures

1 Introduction

- In this section, you will learn where UNDERSTAND fits within the greater goal of understanding intelligence, and the particular problem I use it to solve. You will see a high-level overview of what UNDERSTAND's various components do and how they fit together.
- By the end of this section, you will know enough about UNDERSTAND to describe it to your boss or grandparents. You will also know where to skip ahead to if you want to read more about particular parts of the program.

1.1 The Big Picture: The key to intelligence is representation.

Imagine a dog flying to Mars. This is an unusual thought, and so you may be wondering why I ask you to imagine it. I want you to consider the thought itself.

Is the thought words? When you think of a dog flying to Mars, do the shapes of the letters and words "A dog flying to Mars," come to mind? Not really. If you try, you might be able to conjure up these shapes, but the core thought itself is not linguistic. You produce the words from the thought.

Is the thought a video? If you try you can see it—perhaps you recall a famous photo of Mars, or your own pet—but this imagery is nothing like a movie, and the abstract thought is even less cinematic. Again, your imagination produces these images *from* the thought.

You can find in all our senses this phenomenon of a core thought that imagination can recreate or put into words. We can think about smelling a rose, hearing Miles Davis play, touching denim, or tasting a banana in an abstract way, independent of the senses we first experienced these thoughts through or the words we use to describe them. And we can apply our imagination to the thought to almost smell the fragrance, hear the trumpet, feel the texture, or taste the world's superior fruit. As food and art critics prove, we can also describe our thoughts of smell, sound, touch, and taste in words.

We have many different types of thoughts. "A dog flying to Mars" is a sort of motion through space, a *trajectory*; "the dog becomes excited" is a sort of state change, a *transition*; "the dog is brown and Mars is a planet" are timeless, unchanging *classifications*. We can also think about causality—"NASA made the dog

fly to Mars"—and its opposite, prevention—"but his mother tried to stop him."—both *forces*. And we can think about the relative timing of *events*: "The dog flew to Mars just after his third birthday."

One approach to understanding intelligence is to try to model these abstract thoughts. A representation-centric approach to understanding intelligence focuses on figuring out what pieces of knowledge we need to store in our thoughts, how those pieces relate to each other, and how a computer can store this all efficiently and perspicaciously so it can use it later.

If a computer is going to be as smart as we are, it needs to be able to think about what we can think about. It must have a *representation* for each type of thought that we have; it must have representations to match our *trajectories*, *transitions*, *forces*, etc. (Figure 1) GAUNTLET is a research platform designed to let us represent many types of thoughts by instantiating representations as semantic networks. Around our group we colloquially call these instantiations *frames*, approximating—though not entirely like—the frames envisioned by Minsky.[4]

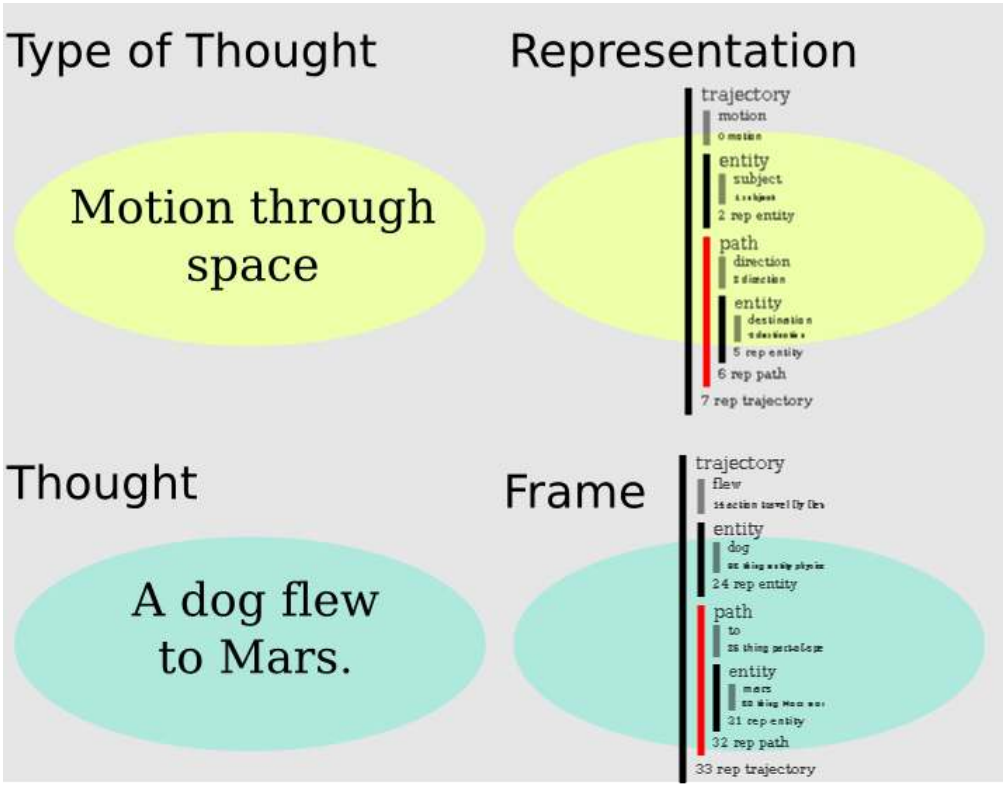


Figure 1: If a thought is a particular instance of a type of thought, a frame is a particular instance of a representation.

AI researchers want frames to work with. We can use them to do analogical reasoning: "That dog is a lot like the Russian space-dog Laika." We can try to make sense out of odd sounding thoughts: "I've never heard of a dog flying to Mars. Perhaps 'dog' is meant to somehow describe a human astronaut."

Given a rich source of useful frames, GAUNTLET can actually reason this cleverly. But it critically needs that source of frames; without frames, GAUNTLET has nothing to reason *on*. UNDERSTAND is that source of frames.

1.2 UNDERSTAND is the third attempt at transforming English into the representations GAUNTLET researchers want to use.

The purpose of UNDERSTAND is to make it easy for GAUNTLET researchers to transform English sentences into frames.

Off-the-shelf tools such as parsers and WordNet[3] already allow a computer to partially understand English.

GAUNTLET first passes your English input into a parser, which analyzes the grammar of your input to mark the language with syntactic information. GAUNTLET also pulls category information from WordNet, such as that "cat" is a type of of mammal. The English, syntactic information, and category information are UNDERSTAND's input. (Figure 2)

UNDERSTAND is the third attempt at writing a computer program for this automatic interpretation. Originally GAUNTLET used a hand-coded rule-based system, but this approach ultimately proved impracticable. LANCE replaced the rule-based system with a program that can learn by example. UNDERSTAND builds on the success of LANCE.

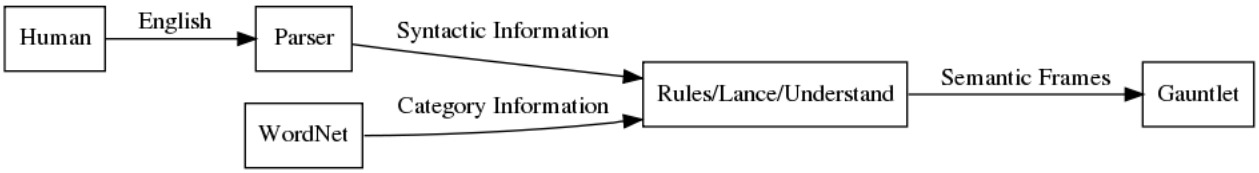


Figure 2: The flow of information into GAUNTLET. First a human types in some English. Then a parser augments the English with syntactic information. UNDERSTAND transforms that syntacticly-augmented English into semantic frames and delivers them to GAUNTLET.

1.2.1 Hand-coding rules is too hard.

The Genesis group's early frame-generation work consisted of manually produced rule-based systems. Human programmers wrote If-Then rules that look for particular English fragments to produce frames: "If you see the parsed form of 'X Ys to the Z', turn that into a *trajectory* frame whose subject is X, motion is Y, and destination is Z." While these rules work well, this manual approach makes it difficult to expand beyond very simple phrases. It would be insane to try to write down all such rules it would take to cover all of English. The manual approach is also distasteful from a cognitive science perspective: we humans learn to understand language by example, not by hard-coding.

1.2.2 LANCE pioneered learning by example.

LANCE [2] pioneered the approach of learning rules by example. It can learn to turn parsed English into frames from a list of examples: "'The dog flew to Mars.' should be represented *this way*, 'The dog became excited.' should be represented *that way*." It uses Winston's Arch-Learning technique[7] to build models for each representation, learning constraints such as "A *trajectory*'s verb must be a type of travel." or "A *transition*'s verb must involve change.".

The purpose of UNDERSTAND is to improve on LANCE by learning equivalent rules from fewer examples.

1.3 Lattice-Learning enables UNDERSTAND to learn from very few examples because it learns abstractions and learns generously.

There are two ways to reduce the number of examples it takes to learn a set of rules:

1. Reduce the number of examples UNDERSTAND needs to learn a single rule.
2. Ensure that the number of examples needed to learn a new rule remains independent of the other rules it depends on.

UNDERSTAND uses a learning technique I call Lattice-Learning to accomplish both of these goals.

To reduce the number of examples it takes to learn any particular rule, Lattice-Learning defaults to very permissive descriptions of what inputs are acceptable. Rather than slowly building up a description from small, sure steps, Lattice-Learning begins by making a huge skyward leap, allowing nearly anything at first, then reigning itself in with future examples. While this style of learning may seem reckless, it learns quickly and the algorithm is designed to prevent mistakes. For most concepts, Lattice-Learning takes just two examples to learn a description: a positive example to seed the description, and a negative example to specify it. For example, Lattice-Learning can learn a description of mammals by answering the question, "What could be a cat but not a fish?"—a positive "cat" example and a negative "fish" example.

UNDERSTAND decouples the task of learning a new rule from old ones with abstraction. By learning each new rule as a "black box", UNDERSTAND can avoid combinatoric blow-up. When learning a rule to produce *trajectories* from *paths*, Lattice-Learning's reasoning is something like, "If I can use *this* path in this *trajectory*, I can probably use *any* path in *any trajectory*."

1.4 UNDERSTAND learns and uses rules in a tightly-coupled loop.

UNDERSTAND embodies to the maxim "You can only learn what you almost already know." Before learning a new rule, UNDERSTAND uses all the rules it has already learned, transforming its English input into frames where possible. Then it learns a new rule based off that nearly-complete transformation.

Whenever UNDERSTAND is learning, the program can be thought to be in a feedback loop of learning and using rules. When it has no rules—no knowledge of English— it can only learn rules for very simple phrases such as "the fish". As its understanding of English grows to a larger list of rules, it can learn new rules based not solely on English, but also on frames produced by rules it learned earlier. This tightly-coupled feedback loop allows UNDERSTAND to get a handle on basic English grammar in a very short period of training. In fact, after learning about a dozen rules, it's possible to teach UNDERSTAND with purely English examples, such as "Patrick's cat means the cat of Patrick."

1.5 Where do I go from here?

- **Supporting Technology** to learn about Threads and Semantic Networks, the representational in-

frastructure of GAUNTLET and UNDERSTAND.

- **Lattice-Learning** to learn more about Lattice-Learning, the reason UNDERSTAND can learn quickly.
- **Rules Engine** to learn how UNDERSTAND uses the rules it has learned to turn English into frames.
- **Results and Contributions** to see my latest experimental results and a summary of this thesis' main contributions.

2 Supporting Technology

- In this section, you will get an overview of *Threads*, *Semantic Networks*, *Dependency Parsers*, *WordNet*, four technologies on which GAUNTLET and UNDERSTAND depend.
- By the end of this section, you will know how each technology fits into UNDERSTAND's architecture, and how it uses each to accomplish UNDERSTAND's goal of learning to understand English.

2.1 Threads store hierarchical data.

Threads[6] are data structures that mimic the way our brains store hierarchical data, such as phylogenetic Is-A relations between different types of animals. (Figure 3)

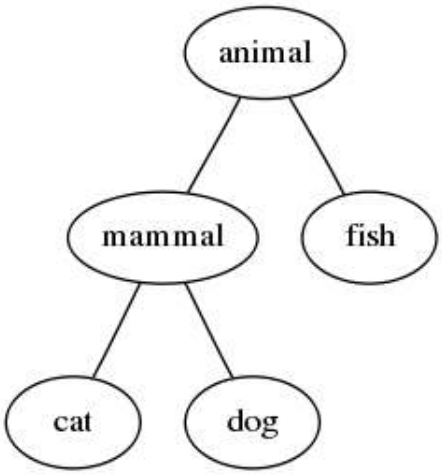


Figure 3: Cats and dogs are mammals. Mammals and fish are animals.

To represent this information as Threads, just write the types of each animal from general to specific. (Figure 4)

```
animal mammal cat
animal mammal dog
animal fish
```

Figure 4: Three Threads express the same information as the tree in Figure 3

The Thread representation of hierarchical memory has several advantages over tree structures.

2.1.1 Advantage 1: Threads store redundant information.

If a tree-based memory were to suffer damage, it would certainly lose information. (Figure 5) If the link between "animal" and "mammal" breaks, there is no way to know that cats or dogs are animals.

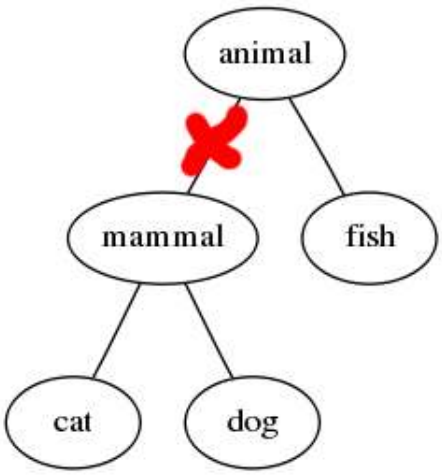


Figure 5: Figure 3 broken. There is no path from "cat" or "dog" to "animal".

On the other hand, the Thread representation of the same memory has redundant information. Again we break one link, between "mammal" and "animal" on the "cat" Thread. (Figure 6)

```
mammal cat
animal mammal dog
animal fish
```

Figure 6: Figure 4 broken. We can fix the "cat" Thread with the redundant information in the "dog" Thread.

There is still enough information in the Thread Memory to determine that cats are animals: reading the "dog" Thread, we see that mammals are animals. Cats are mammals, so we can deduce that cats are animals.

2.1.2 Advantage 2: Threads are biologically plausible.

Vaina and Greenblatt, the creators of the Thread Memory representation, showed that damage to Thread Memory mimics some types of aphasic memory loss. That is, a damaged Thread Memory shows some of the same disabilities as a damaged human brain. This parallel suggests that human memory may be implemented in a Thread-like way.

2.1.3 Advantage 3: Threads easily handle multiple word senses.

When we hear the words "dog" and "fox", we typically recall their most common meanings, the common animals. But these words have several more meanings: to call a person a dog means you consider them ugly; to call someone a fox means you consider them clever. Thread Memory can represent these different word senses with a different Thread for each. (Figure 7)

```
entity living-thing animal mammal carnivore canine dog
entity living-thing person unpleasant-person unpleasant-woman frump dog
entity living-thing animal mammal carnivore canine fox
entity living-thing person clever-person fox
```

Figure 7: Four Threads: two senses of the word "dog" and two senses of the word "fox".

GAUNTLET uses Threads as unique identifiers for a concept. Thread Memory's handling of multiple senses allows precision: instead of reasoning about the words "dog" or "fox", GAUNTLET reasons about Threads as particular word senses. Threads remove ambiguity: they tell us whether the word "Mike" refers to me, to some other person named Mike, or to a microphone.

In classical AI terms, Threads are like symbols, but contain richer type information. Unlike a meaningless LISP gensym, Threads contain information about how the symbol relates to other symbols.

2.2 Semantic Networks store relations between threads.

While GAUNTLET uses threads to represent one particular concept, such as Luke Skywalker or Darth Vader, threads are not suited for storing relationships *between* concepts, such as that Vader is Luke's father. GAUNTLET uses the Semantic Network data structure to store inter-concept relationships.

2.2.1 Simple Semantic Networks store relations between concepts.

Semantic networks are typically drawn as graphs, with concepts as nodes and relationships as edges.

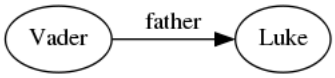


Figure 8: Vader is Luke's father.

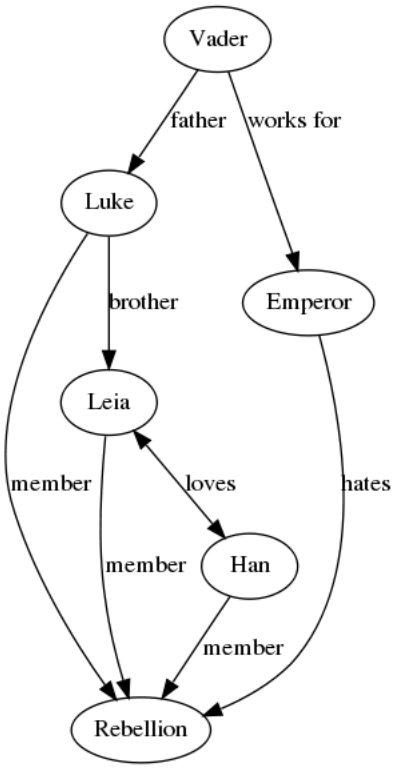
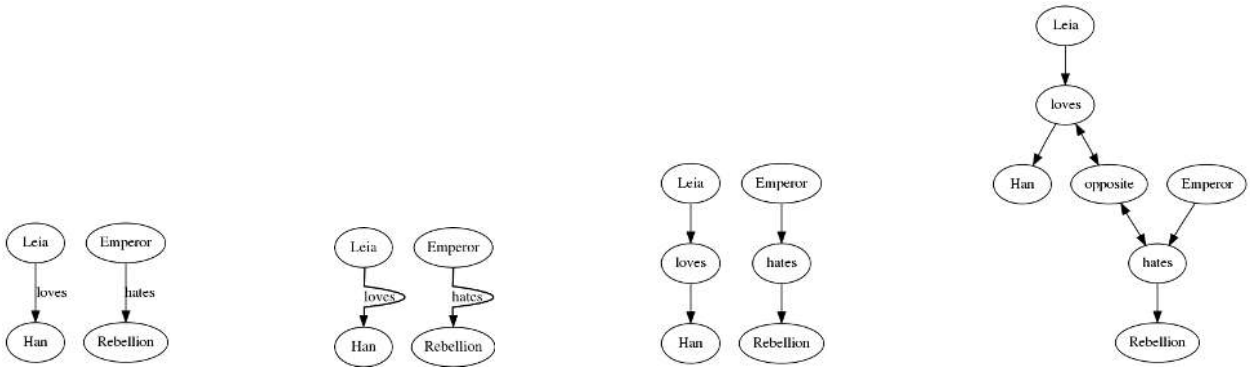


Figure 9: A richer version of Figure 8 with more Star Wars relationships.

2.2.2 Reified Semantic Networks store relationships between concept and relationships.

Unfortunately these simple semantic networks are a bit limiting. Though we can use them to express the relationships between any two objects, we cannot use these semantic networks to talk about relationships themselves. We can say "Leia loves Han." and "The Emperor hates the Rebellion." but we cannot express "loves is the opposite of hates".

A reified semantic network is called so because it turns relationships into things (Latin: *res, rei*) themselves. It is easiest to see how this type of semantic network differs from a simple semantic network graphically. (Figure 10)



(a) Leia loves Han and the Emperor hate the rebellion.

(b) Imagine hooks pulling on the relationship lines to make loops of their own.

(c) The loops become nodes representing each relationship.

(d) Now we can draw the opposite relationship between "hates" and "loves"

Figure 10: Edges in a simple semantic network become nodes in a reified semantic network.

2.2.3 GAUNTLET uses reified semantic networks as a meta-representational substrate.

GAUNTLET builds frames out of four primitive semantic network combinators: things, relations, derivatives, and sequences. *Derivatives*, *relations*, and *sequences* can combine any of the four combinators. This flexibility allows GAUNTLET to build arbitrarily complex representations.

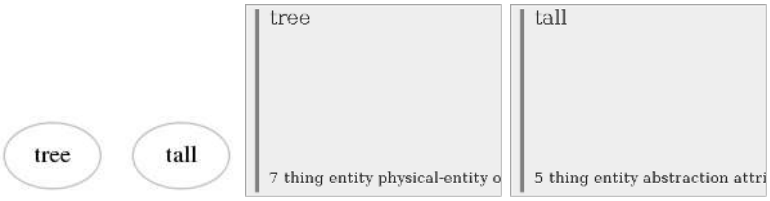


Figure 11: *things* are containers for threads. *things* are like the nodes of a simple semantic network: they represent a single concept.

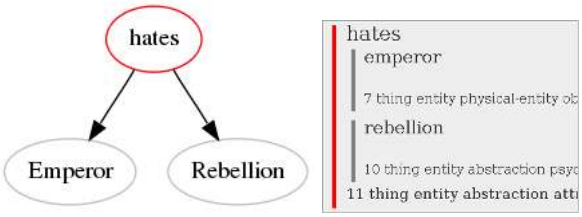


Figure 12: *relations* connect two concepts together, as the edges of a simple semantic network do. Like *things*, *relations* have a descriptive thread.

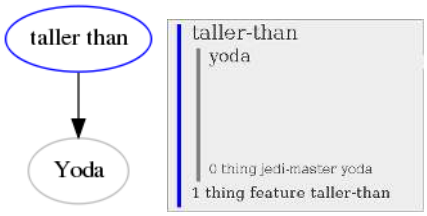


Figure 13: *derivatives* derive meaning from one thing. *derivatives* are like a one-sided relation.

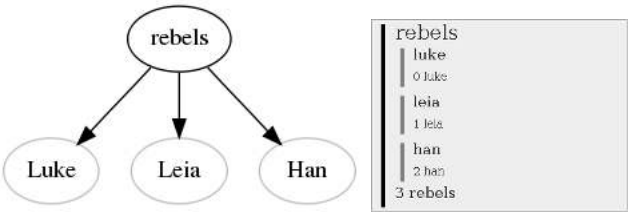
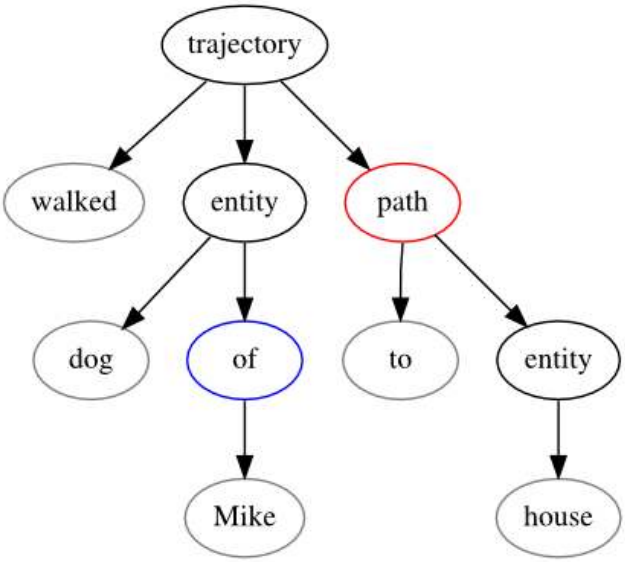


Figure 14: *sequences* gather together any number of things. *sequences* are like a joint relation between many concepts.



trajectory
- walked
  - 10 action travel walk walked
- entity
  - dog
    - 19 thing entity physical-entity object whole living-thing organism animal chordate vertebrate mam
  - of
    - mike
      - 20 thing person mike
    - 21 thing preposition of
  - 22 rep entity
- path
  - to
    - 23 thing preposition to
  - entity
    - house
      - 38 thing entity physical-entity object whole artifact structure housing dwelling house
    - 39 rep entity
  - 40 rep path
- 41 rep trajectory

Figure 15: Using the four basic combinators, it is easy to construct frames that begin to capture aspects of meaning, such as this one representing "Mike's dog walked to the house."

2.3 Dependency Parsers diagram sentences.

A dependency parser is a program that can turn English text into a set of simple syntactic relations. Running a dependency parser on a sentence results in something much like a grade-school sentence diagram. UNDERSTAND uses a dependency parser as the first step in turning English into the representations you teach it.

UNDERSTAND mechanically transforms a dependency parse into a set of GAUNTLET frames. It creates one relation for every pair of related words in the sentence.

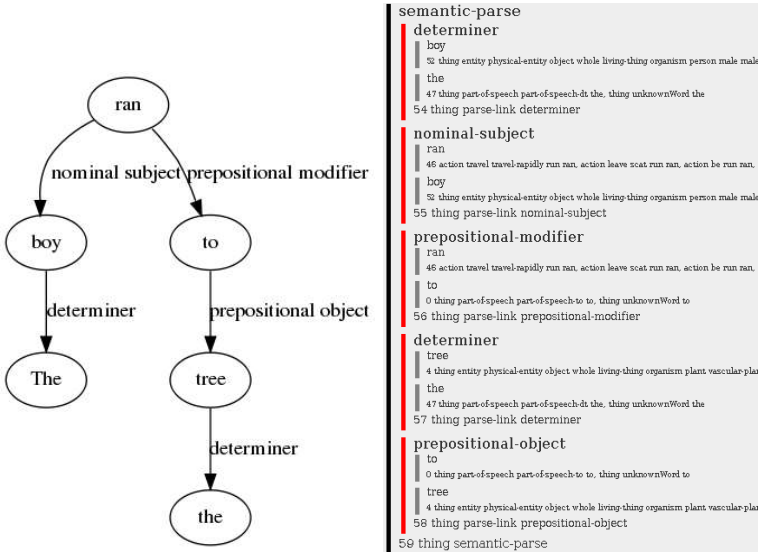


Figure 16: The dependency parse of "The boy ran to the tree."

UNDERSTAND currently uses the Stanford Parser[1] as its dependency parser. But the parser is a modular component of UNDERSTAND, and could be replaced with any other English parser with little modification.

2.4 WordNet is GAUNTLET's Thread dictionary.

WordNet is a database of relationships between English words. WordNet stores relationships such as *synonym*, *antonym*, and importantly for GAUNTLET, *hyponym*. Hyponym relationships contain Thread-like information: "cat" is a hyponym of "mammal."

GAUNTLET imports WordNet's hyponym information to build Threads.

3 Lattice-Learning

- In this section, you will learn about a new algorithm for concept learning, Lattice-Learning. You will see that its generous descriptions allow it to learn quickly.
- I will use a *mammal* description as a motivating example. In particular, you will see what you can learn about mammals by knowing cats are mammals and fish are not. I will then show a real example of how UNDERSTAND learns descriptions to match *entity* and *path* representations.
- After reading this section, you will have learned a bit about how Lattice-Learning fits into the larger category of concept-learning algorithms. You will know how to rank concept-learning algorithms on a scale between suspicious and generous. You will know why generous algorithms like Lattice-Learning are a good fit for learning rules to transform English into frames.

Lattice-Learning is an algorithm for concept-learning on Threads. You can use Lattice-Learning to build a description of a concept from positive and negative examples.

3.1 Concept-learning algorithms learn descriptions from examples.

To learn a concept, a computer program tries to determine a Boolean function that tells whether or not an unknown input is in that concept. For example, if the concept to learn is *mammal*, then

$$mammal(cat) = True$$

$$mammal(fish) = False$$

That is, cats are mammals but fish are not.

To teach the program, we just have to feed it such example cases of the function, and the learning algorithm infers the rest. How exactly the algorithm infers the rest defines a concept-learning algorithm.

We can think of concept-learning algorithms as narrowing an upper and lower bound in a Version Space[5]. (Figure 17) Learning a better description moves the two bounds closer together.

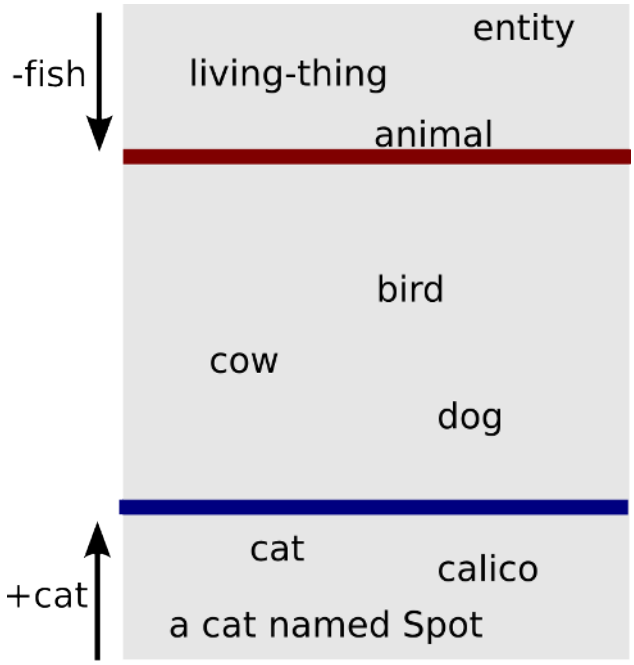


Figure 17: A version space for *mammal* after $+cat, -fish$. Too-general types such as *living-thing* are above the upper bound, and known-mammal types such as *cat* are below the lower bound. *dog* remains in the uncertain middle area.

The lower bound marks types certain to be in the concept; we know that cats are in *mammal* because we explicitly told the program so. *cat*, as a positive example, generalized the description from nothing to at-least-cats.

The upper bound cordons off too-general types; we know that *living-thing* is not within *mammal* because we know fish are living things but not mammals. $fish$, as a negative example, specified the description from anything down to nothing-that-could-possibly-be-a-fish.

It is not obvious what to do with an input like *dog*, which falls in the area between the lower and upper bounds. It is not a cat, but also could not be a fish: no example explicitly allows dog, and no example denies it. How to treat this middle area is a design choice in a concept-learning algorithm.

Winston's Arch-Learning algorithm at the heart of LANCE denies anything in this middle area: its *mammal* function would return $False$ for a *dog* input. To understand that dogs are mammals too, we must guide LANCE with more examples to climb up the list of types from cat to mammal. LANCE's suspicion prevents it from learning quickly, demanding many examples before learning a concept fully.

Lattice-Learning, on the other hand, treats the middle area as $True$. Its reasoning mirrors the human logic, "Will I allow it? Sure, I don't see any reason why not to." This generosity enables UNDERSTAND to settle on the correct *mammal* description from only the cat and fish examples.

3.2 Lattice-Learning learns *mammal* from just $+cat$ and $-fish$

To learn a concept, Lattice-Learning keeps a record of all the positive and negative example Threads of that concept it has seen. At any time it can use these positive and negative examples to generate a description of the concept on-the-fly.

To generate a description, Lattice-Learning tries to answer the question "What are the most general types of things that could be one of my positive examples but cannot be one of my negative examples?" In this section I step through the mammal example to show how the algorithm computes the answer to that question. Our examples are $+cat$ and negative $-fish$.

The algorithm walks down the types on each positive example Thread from general to specific. These types are candidates to be in the set of most general types. In this case, there is just one positive Thread, *cat*, beginning with *thing*.

```
thing entity physical-entity object whole living-thing organism animal chordate
vertebrate mammal placental carnivore feline cat
```

Figure 18: Just one positive Thread, *cat*.

The algorithm walks down the types on each candidate thread, checking each type to see if it is in any of the negative threads. If the type is part of a negative example, the type is too general; if the type is not part of any negative example, it is safe to admit. Because the algorithm works from general to specific, the first type that is safe to admit is the most general.

I'll include a few more examples. Note that $+cat$ allows anything, as it has no reason not to—no negative examples.

- "Can thing be a cat but not a fish?" No, a fish is a thing, so some unknown thing could actually be a fish.
- "Can entity be a cat but not a fish?" No, again, an unknown entity could be a fish.

⋮

- "Can a vertebrate be a cat but not a fish?" No, fish are vertebrates.
- "Can a mammal be a cat but not a fish?" **Yes**, an unknown mammal could be a cat, but cannot be a fish, as fish are not mammals.

Figure 19: A trace of the algorithm deciding that *mammal* is the most general type allowed by $+cat, -fish$.

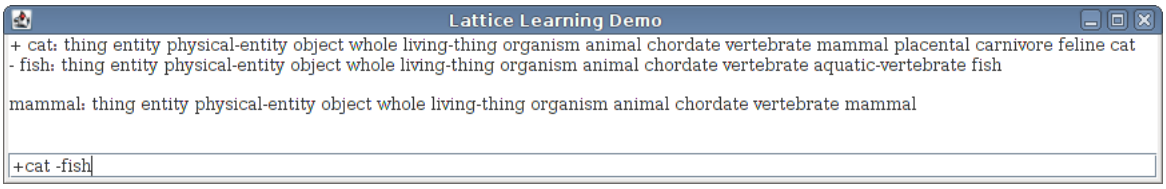


Figure 20: A screen-shot showing the implemented Lattice-Learning code also settling on *mammal* from $+cat, -fish$.

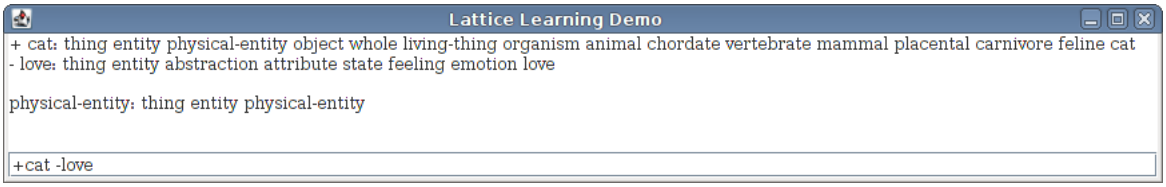


Figure 21: $+cat, -love \rightarrow physical\text{-}entity$

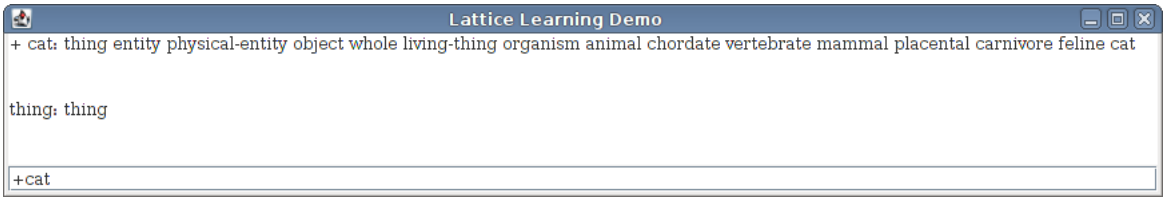


Figure 22: $+cat \rightarrow thing$

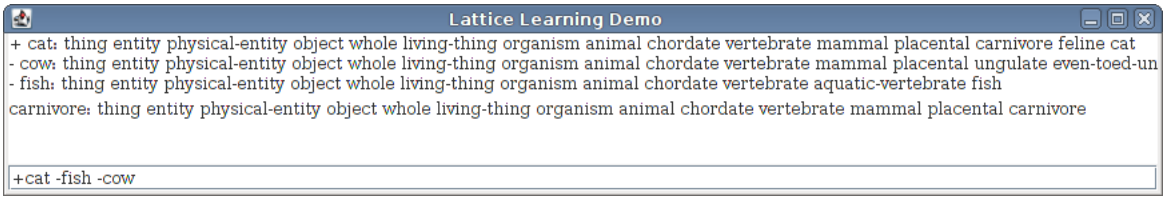


Figure 23: $+cat, -fish, -cow \rightarrow carnivore$

3.3 Why not just tell the program "mammal" to begin with?

You may be wondering why we have to go through such a hassle to teach the computer a description for mammal. Could we not just have given it mammal to begin with? Yes, in this case we certainly could have. But we can only do so *if we know what the description should be.*

One particularly fascinating aspect of the human mind is that it is sometimes much easier for us to come up with specific examples than a general description. Sometimes we do not even have a name for the general description we want to consider. It is much easier for us to say "a dove, robin, and sparrow are this type of bird, but hawks, penguins, and ducks are not" than to remember the order of small, perching songbirds, "passerines." Lattice-Learning shines when we want to teach a concept whose description we cannot name explicitly.

Because Lattice-Learning can generate descriptions on-the-fly, it is well-suited to refinement over time. Perhaps originally you thought you were teaching the program about *mammals*, but upon reflection you realize you meant *carnivores*. Just add *cow* as another negative example. (Figure 23) Lattice-Learning automatically refines its description to account for every example. And because it remembers all examples and generates a fresh description with each refinement, Lattice-Learning is not sensitive to what order you give it the examples.

3.4 UNDERSTAND learns descriptions of frames

Until now I have only shown Lattice-Learning working with a single description whose positive and negative examples are threads. UNDERSTAND extends this learning to *trajectories* and other frames—structures composed of threads.

The first representation I usually teach UNDERSTAND is *entity*, a way of storing a noun and all its related information together; "the dog", "cats", "Patrick's pet fish", and "the very ugly red dog" are all *entities*. So I start by telling UNDERSTAND the meaning of "the cat." (Figure 25)

Applying Lattice-Learning at each level of the input, the description to match the input looks like Figure 25.

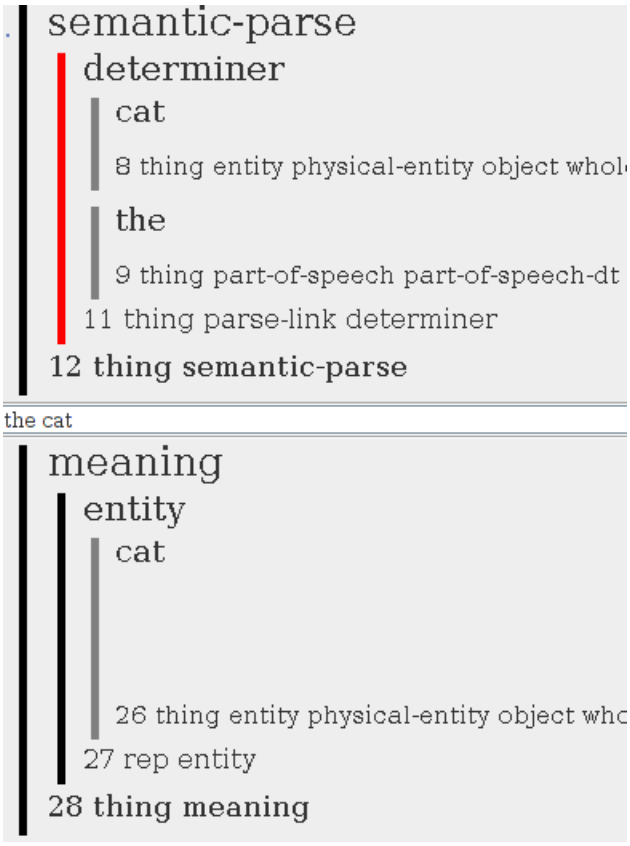
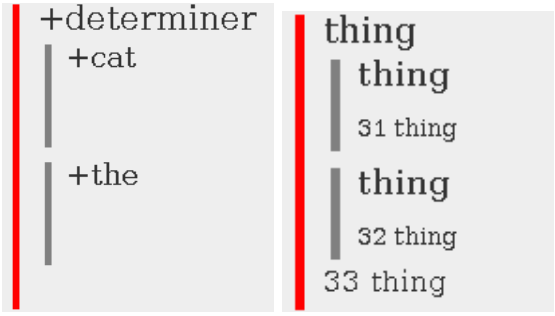


Figure 24: I tell UNDERSTAND to represent "the cat" as an *entity*.



(a) Positive and negative examples

(b) Generated description

Figure 25: Each thread is now a positive example. So far, this description has no negative examples. This description will match any relation with two things underneath.

But this description is too general. A side-effect of the Lattice-Learning algorithm is that a description consisting of just a single positive example and no negative examples will match anything. In particular, we don't want it to match the "to tree" part of "to the tree." (Figure 26)

To fix this problem, we let UNDERSTAND make the mistake, and then point it out as a negative example. (Figure 27)

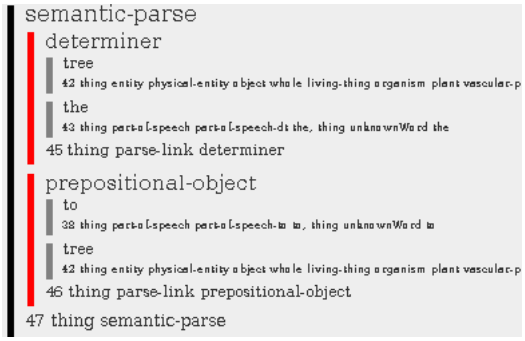


Figure 26: The parse of "To the tree." is two relations. According to our current description, both are allowed. That is bad. We want "the tree" but not "to tree".

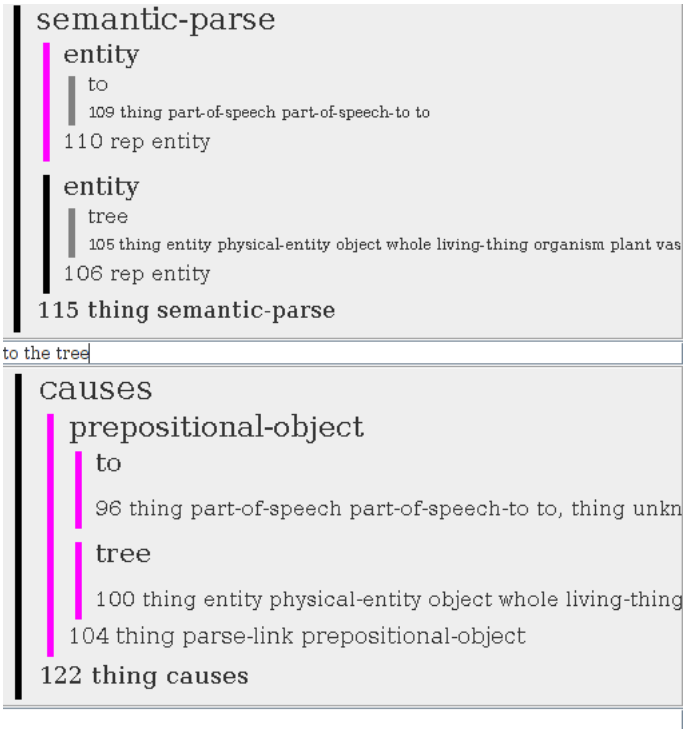


Figure 27: I tell UNDERSTAND to fix the description with a negative example. I mark the errors in pink by clicking on them.

(a) Positive and negative examples (b) Generated description

Figure 28: The description now matches only a determiner relation between an entity and a part-of-speech. This is probably specific enough.

3.5 UNDERSTAND learns abstractions

The rule that produces *entities* dealt entirely with English syntax—the *entity* representation does not incorporate any other representations. But UNDERSTAND can build representations out of both syntax descriptions and other representations. This process is analogous to building walls out of basic components like bricks (syntax), and then building houses out of already-built walls (other representations).

To see how UNDERSTAND uses other representations, consider what the parse of "to the tree" looks like now, having learned the right rule for *entity*.

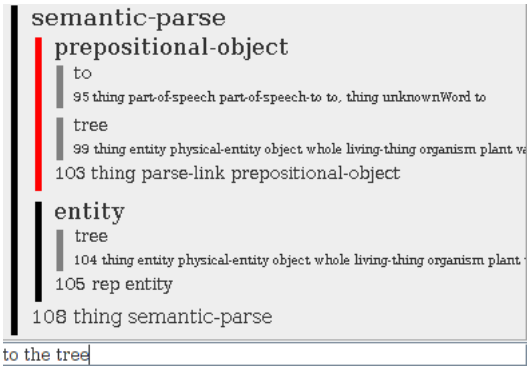


Figure 29: The parse has one syntax relation, and one *entity*.

I usually represent *paths* as a relation between the preposition and the *entity* defining the path.

UNDERSTAND treats this case a little differently. Noting that it pulled in the whole *entity* untouched, UNDERSTAND sees an opportunity for abstraction. Instead of building a complete description, it treats *entity* as a black box, only building a description at the very top level.

Because UNDERSTAND treats the *path*'s internal *entity* as an abstraction, the rule does not have to change if we expand on the *entity* representation later. If we teach UNDERSTAND how to turn "Mike's house" or "the top of the tree" into *entities*, "around Mike's house" and "from the top of the tree" will already work.

Figure 30: We want the parse to turn into this. Note that the *entity* has been swallowed whole.

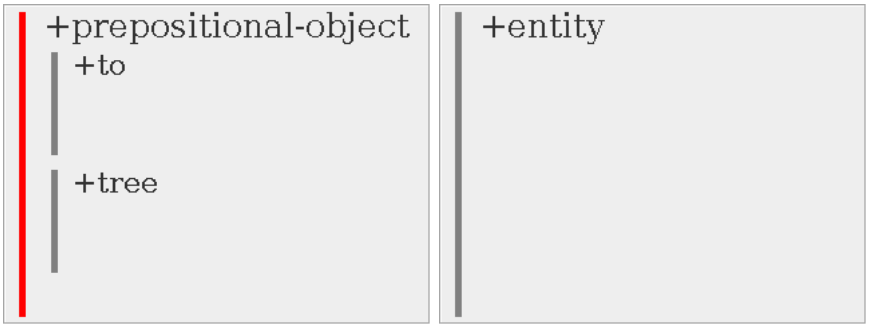


Figure 31: The syntax relation has a full description, but the description for the *entity* is shallow. This rule does not care what is inside the *entity*.

4 Rules Engine

- In this section, you will learn how UNDERSTAND's engine for using rules works. You will see how it uses the rules it has learned to turn simple frames into fewer, richer frames. And you will learn how the *Deepest-Average heuristic* and *priming* ensure that you get back the interpretation you want when the meaning is ambiguous.
- I'll first show how the engine transforms the simple phrase "to the tree" into a path frame. To show how the engine handles ambiguity, I'll also show what it does with "The bird soared over the sea."

UNDERSTAND learns rules that fit nicely into a rewriting rule system. UNDERSTAND receives examples from the user in the form of an example input and the desired output for that input. When UNDERSTAND learns a new rule from an example, it builds Lattice-Learning descriptions based on the example input, and it builds skeletal versions of the example output. (Figure 32)

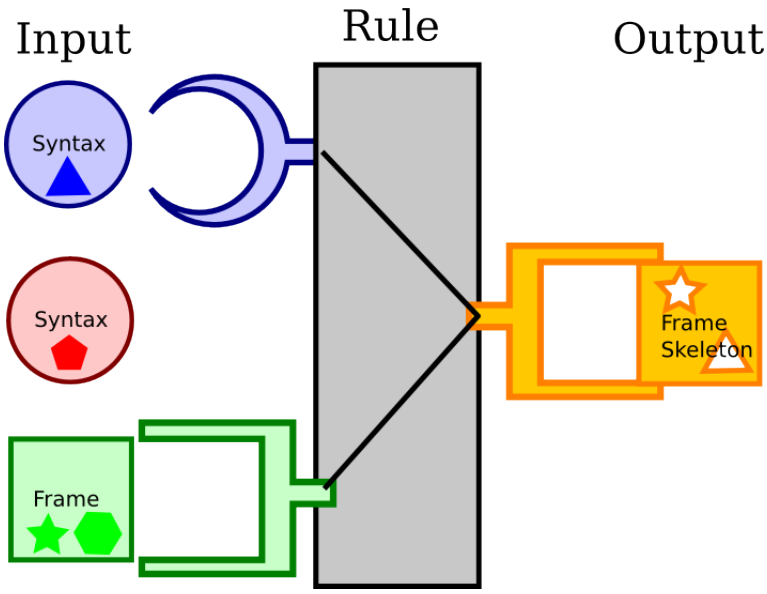


Figure 32: A stylized rule. On the left side, a rule looks for syntax or frames to match each of its descriptions. If all the rule's descriptions are matched, the rule produces a new frame on the right by filling a skeleton.

The rule's skeletons have the same shape as the example's output, and know how to fill themselves in. The skeleton stores some combination of constant, already-filled-in Threads, paths to find a Thread in the input, or paths to find a whole frame in the input.

The skeleton stores constant Threads for Threads it cannot find in the input, such as `rep path` when learning to turn "to the tree" into a *path*, as in Figure 30.

If a Thread is in the input, such as `thing entity ... tree` in the same example, the skeleton stores an encoded path to that Thread: a frame number in brackets followed by a list of turns to make while descending into that frame. Frame numbers count from zero, so you may need to add 1 to each number to make sense of the path. For example, `[2] 1 0` means: look into the frame in the input that matches the third description; then step into its second component; step into *its* first component; finally get the Thread of that innermost part. (Figure 33)

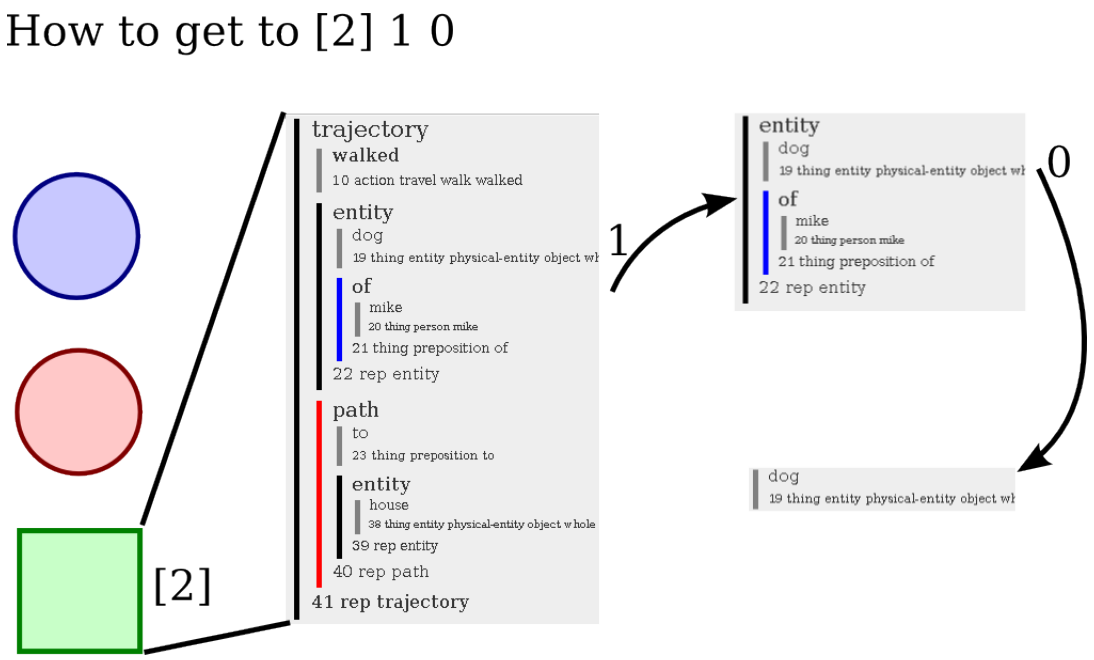


Figure 33: The thread at `[2] 1 0` in this example is *dog*.

If a complete frame from the input is found in the output, as the *entity* frame, the skeleton stores a path indicating it can pull the whole frame. Whole-frame paths are encoded as a frame number in braces: {3} means to pull in the entire fourth frame. This path encoding needs no more numbers, as the skeleton does not descend into the frame.

A collection of input descriptions and skeletons make a rule. If the input descriptions are satisfied by the input frames, the skeletons fill themselves in to produce new output frames.

Though production systems all use rules in roughly the same way, they differ in what to do if more than

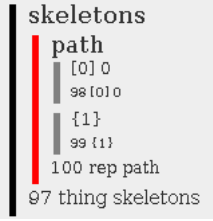


Figure 34: A skeleton with all three types of skeleton options: `rep path` is a constant Thread; `[0] 0` points to the Thread of the first frame's first component and {1} points to the entire second frame.

Figure 35: Rule #0: The rule for creating an *entity* UNDERSTAND learned in the last section looks for the particular syntax of a noun phrase. It creates an *entity* by pulling in the Thread for the noun.

one rule can fire at any time. UNDERSTAND's engine takes a multiple-universes approach: when several rules apply, it follows each possibility independently. Once each universe has settled on an answer, it chooses the best result to return as its interpretation of the input. This implementation naturally allows priming—telling the engine in advance what kind output you think is the best.

4.1 Normally only one rule applies at a time.

Consider an unambiguous sentence fragment, "to the tree." (Figure 37)

On an unambiguous example like "to the tree," only one rule applies at a time. The rule engine simply applies each rule as it becomes applicable. When no more rules apply, the rule engine stops and returns

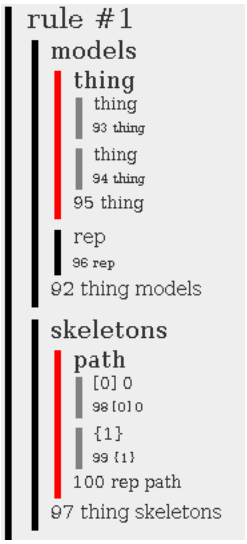


Figure 36: Rule #1: The rule for creating a path. One description matches the destination *entity*, and another looks for the preposition. If both descriptions are satisfied, the skeleton can pull the preposition and entity together to form a *path*.

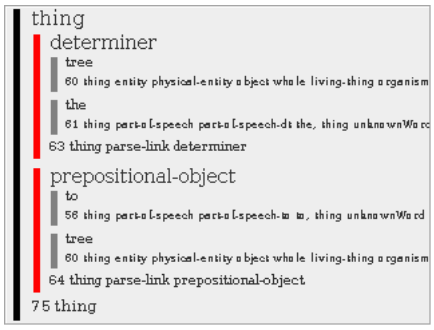


Figure 37: The dependency parse of "to the tree" has 2 links.

the last-produced frames as the final result. (Figure 38) To interpret "to the tree" as a *path*, UNDERSTAND needs just the two rules it learned in the previous section.

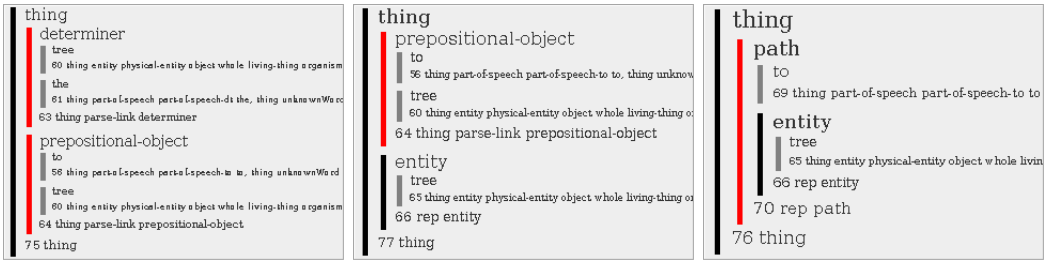


Figure 38: At first Rule #0 is satisfied and Rule #1 is not, so Rule #0 fires. Then, Rule #1 is satisfied and Rule #0 is not, so Rule #1 fires. After these two steps, no rule is satisfied, so we're done.

4.2 Ambiguity forces UNDERSTAND to explore all possibilities.

Sometimes UNDERSTAND is asked to transform a sentence that seems ambiguous. That is, during the transformation of the sentence, UNDERSTAND finds two or more rules that could apply at the same time. One such sentence is "The bird soared over the sea."

"Soar" has always been a problematic word for GAUNTLET researchers, as usually it means "fly", but can also mean "increase", as in "the price soared." The "fly" interpretation leads to a motion-through-space *trajectory*, while the "increase" sense is best thought of as a state change *transition*. While both interpretations of this sentence make some sense, a *trajectory* is a more natural representation, as it can incorporate all the sentence's information. The *transition* representation we use in GAUNTLET cannot incorporate *paths* like "over the sea."

When UNDERSTAND tries to transform a sentence like this, it does so by exploring all possible interpretations. When it reaches the ambiguity, it forks off into two branches: one leading to the *trajectory* interpretation, one to the *transition* interpretation. After each branch finishes, UNDERSTAND passes all possible interpretations to the Deepest-Average heuristic to choose the best one.

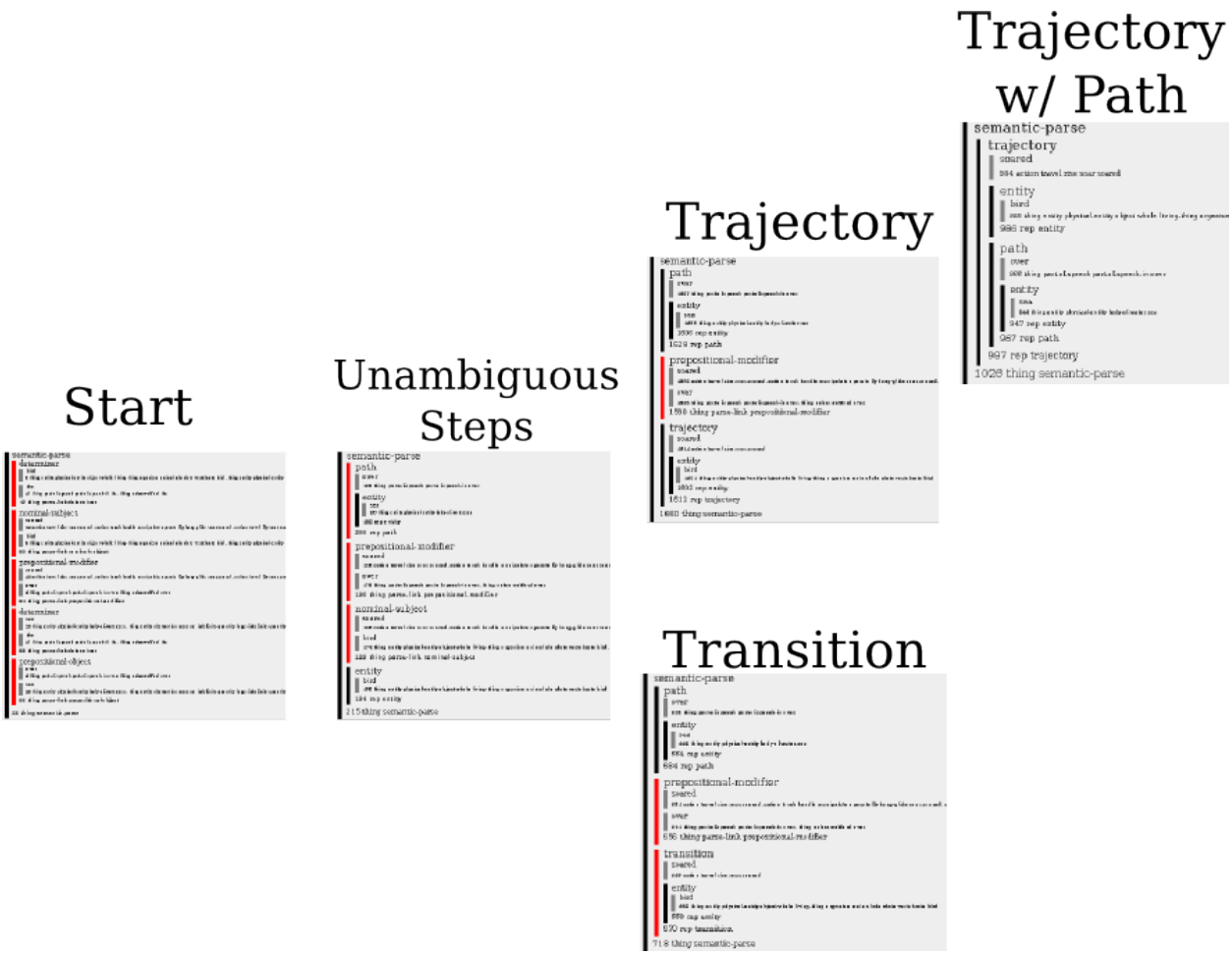


Figure 39: How UNDERSTAND interprets "The bird soared over the sea." First, three unambiguous steps transform "The bird" and "the sea" into *entities*, and then "over the sea" into a *path*. Next, the engine branches into two paths, one interpreting "The bird soared" as a *trajectory*, the other as a *transition*. Finally, the *trajectory* branch incorporates the "over the sea" *path*; the *transition* cannot use the *path*.

4.3 The Deepest-Average heuristic usually chooses the frame you want.

In our first example, "to the tree", there was no ambiguity about what frame we wanted back; we wanted the final one, the *path*. Typically this ultimate product will be the most meaningful.

But when the engine runs into ambiguity and forking paths appear, final frame is an idea that no longer makes sense. In our second example, is the *trajectory* or *transition* the final one? Both are at the end of a branch of execution, so that criterion doesn't help us decide. In these ambiguous situations, we can apply the Deepest-Average heuristic.

To find the Deepest-Average value for a set of frames, calculate the average depth of its frames, counting syntactic parse-link frames as depth 0. Choose the frame set with the largest heuristic value as the best interpretation. You can see the last example from Figure 39 labeled with Deepest-Average values in Figure 40.

Empirically I find that the Deepest-Average heuristic almost always gives back the result I find most meaningful. Looking for a high average depth promotes higher-level interpretations. Counting syntactic parse-links as 0 depth penalizes interpretations that do not use all the available material, as in the *transition* interpretation of "The bird soared over the sea," which did not use the "over the sea" path information.

4.4 If you prime UNDERSTAND with what you want, it will try to find a frame that matches.

Occasionally the Deepest-Average heuristic doesn't give the answer you want; sometimes we ask it to serve two masters. Say Mark wants to find *trajectories* in stories, and Raymond wants to extract *transitions*. Someone will be disappointed by the heuristic's answer. So UNDERSTAND allows you to prime its Deepest-Average heuristic with a description to match.

Primed with a description, UNDERSTAND will apply Deepest-Average only to outputs that match the description, giving back the best of a subset of all possible answers. Mark primes UNDERSTAND with a *trajectory* description, Raymond primes it with a description that matches *transitions*, and both get back the interpretation they are looking for.

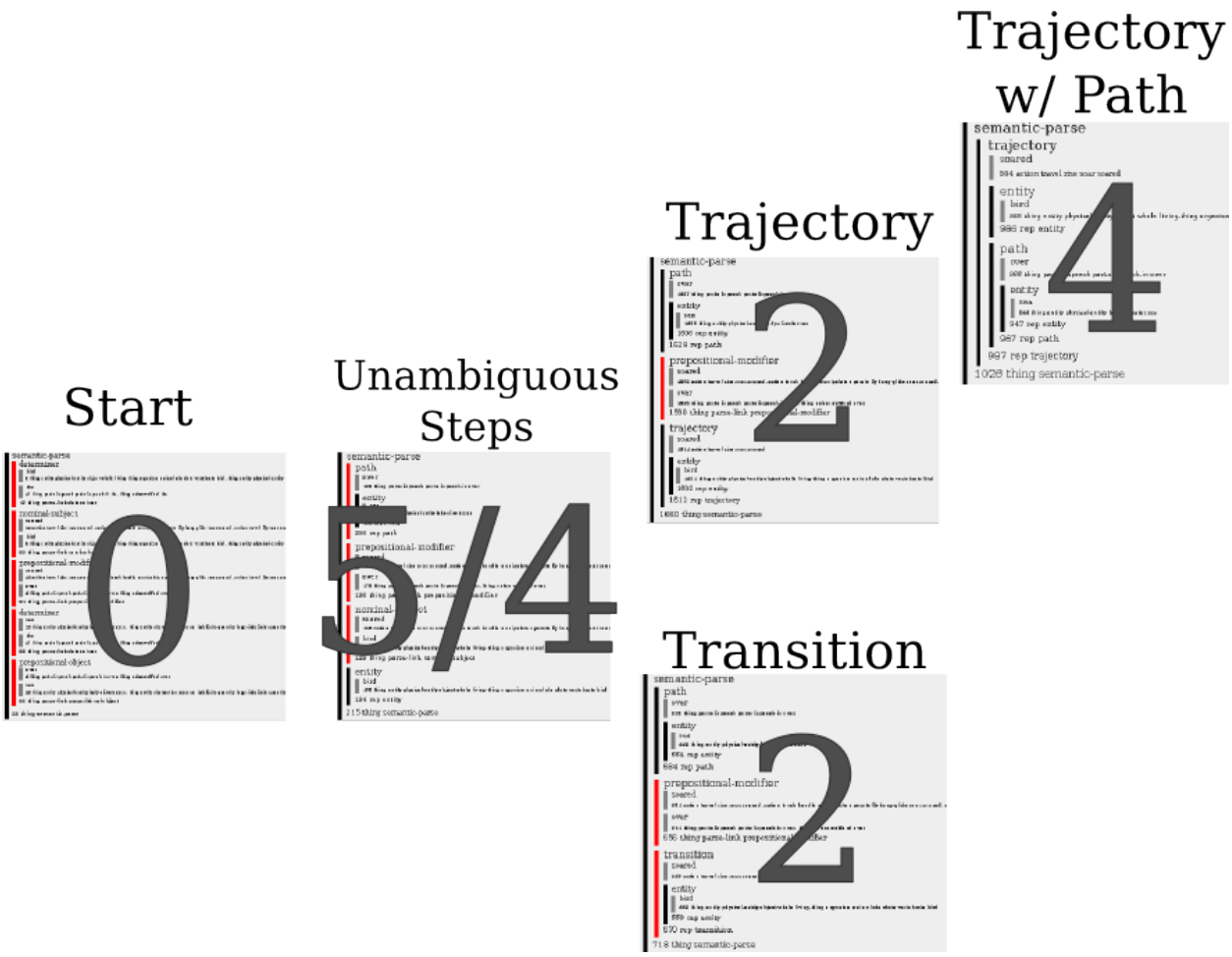


Figure 40: "The bird soared over the sea." labeled with Deepest-Average values. The *trajectory* with a *path* has the best heuristic value.

5 Results and Contributions

5.1 Experimental Results

UNDERSTAND's ability to transform parsed English into semantic frames depends entirely on its training. While we have just begun to explore how much UNDERSTAND will ultimately learn, I provide a sample training sequence for reference and comparison to previous programs such as LANCE. The exact training data can be found an appendix to this document.

In 20 minutes I trained UNDERSTAND with 8 positive and 4 negative examples of what various phrases of English mean (or do not mean). With just these 12 examples, UNDERSTAND can produce frames covering several areas of the English language: entities, paths, trajectories, transitions, causality, and time relations.

5.2 Contributions

In this section, I list my noteworthy contributions in this thesis.

- Introduced Lattice-Learning, a concept-learning algorithm that can learn Thread descriptions from very few examples.
- Implemented UNDERSTAND, a program that uses Lattice-Learning to learn to transform parsed English into semantic frames.
- Suggested that the Deepest-Average heuristic will usually pick the most meaningful of several ambiguous interpretations.
- Performed an experiment showing UNDERSTAND's remarkable grasp of English after only 12 examples.

Appendix: Examples

In this appendix I list the examples that teach UNDERSTAND to the point referred to in my experimental results. These examples are stored in text format.

Positive Examples

Positive examples consist of a `Positive` marker followed by some English text and its meaning. Its meaning can either be in a form of s-expression markup or in English that UNDERSTAND already can understand (from previous examples.) For example,

```
Positive
Mike's car
the car of Mike
```

will teach UNDERSTAND that "Mike's car" means whatever "the car of Mike" means.

Negative Examples

Negative examples are a little trickier. They start with a `Negative` marker followed by some English *context*, an incorrect frame, and faulty causes. Rather than just asserting "this does not mean that", a negative example asserts "in this context, this frame is not a valid interpretation, because these causes prevent it from being so." The second example states that "to" is not an entity in "to the tree" because it is part of a prepositional-object grammar link.

Negative examples are tricky to write manually. Even I shy from it. The format is really tuned for UNDERSTAND; it is easy to work with UNDERSTAND to create these negative examples by marking the mistakes it makes.

S-expression markup

S-expressions are a convenient format for expressing tree data structures as text. It is beyond the scope of this appendix to explain them fully, but I do want to supply a few notes for people familiar with them.

The 't', 'd', 'r', and 's' atoms begin a new *thing*, *derivative*, *relation*, or *sequence*. These expect a thread, followed by 0, 1, 2, or any number of children, respectively.

Threads can either be written literally in quotations, as in `"rep entity"`, or just as a name that will be looked up in WordNet, as in `bird`.

The '*' atom shortcuts the tree building process by pulling in an entire frame, rather than just a thread. It looks for a frame with the thread at its top level, and pulls it into the s-expression whole. `(* entity)`, for

example, pulls in a whole *entity*, in the third example the tree *entity*. This feature is purely for convenience, and has the same effect as typing the complete structure.

The examples

These is UNDERSTAND's exact log file for the training that produced the documented results, with minor formatting to fit the page. There are 8 positive and 4 negative examples.

```
Positive
the bird
(s "rep entity" (t bird))

Negative
to the tree
(S "rep entity" (T "thing part-of-speech part-of-speech-to to"))
(R "thing parse-link prepositional-object" (T "thing part-of-speech
part-of-speech-to to") (T "thing entity physical-entity object whole
living-thing organism plant vascular-plant woody-plant tree"))

Positive
to the tree
(r "rep path" (t to) (* entity))

Positive
the bird flew
(s "rep trajectory" (t flew) (* entity))

Negative
the bird flew to the tree
(S "rep trajectory" (T "action travel fly flew") (R "rep path" (T "thing
part-of-speech part-of-speech-to to") (S "rep entity" (T "thing entity
physical-entity object whole living-thing organism plant vascular-plant
woody-plant tree"))))
(R "thing parse-link prepositional-modifier" (T "action travel fly flew") (T
"thing part-of-speech part-of-speech-to to"))

Negative
the bird is
(S "rep trajectory" (T "action be is") (S "rep entity" (T "thing entity
physical-entity object whole living-thing organism animal chordate vertebrate
bird")))
(T "action be is")

Positive
the bird flew to the tree
(s "rep trajectory" (t flew) (s "rep entity" (t bird)) (* path))

Positive
the price increased
```

```
(r "rep transition" (t increased) (* entity))

Negative
the price is
(R "rep transition" (T "action be is") (S "rep entity" (T "thing entity
abstraction attribute quality worth value monetary-value price")))
(T "action be is")

Positive
the man disappears
(r "rep transition" (t disappears) (* entity))

Positive
a star appears
(r "rep transition" (t appears) (* entity))

Positive
the bird flew because a cat appeared
(r because (* transition) (* trajectory))
```

References

[1] Dan Klein and Christopher D. Manning. Fast exact inference with a factored model for natural language parsing. In *Advances in Neural Information Processing Systems 15 (NIPS 2002)*, pages 3–10. MIT Press, 2003.

[2] Adam Kraft. Learning, using examples, to translate phrases and sentences to meanings. Master's thesis, MIT, 2007.

[3] George A. Miller. Wordnet 3.0, 2006. http://wordnet.princeton.edu/.

[4] Marvin Minsky. A framework for representing knowledge. 1974.

[5] Tom Mitchell. *Machine Learning*, chapter 2. McGraw Hill, 1997.

[6] Lucia Vaina and Richard Greenblatt. The use of thread memory in amnesic aphasia and concept learning, 1979.

[7] Patrick Winston. *Learning Structural Descriptions from Examples.* PhD thesis, MIT, 1970.