Word Sense Disambiguation Through Lattice Learning

by

Eli Stickgold

B.S., Electrical Engineering and Computer Science, Massachusetts Institute of Technology (2010)
B.S., Brain and Cognitive Sciences, Massachusetts Institute of Technology (2010)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2011

Author .....
Department of Electrical Engineering and Computer Science
February 1st, 2011

Certified by .....
Patrick H. Winston
Professor
Thesis Supervisor

Accepted by .....
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Word Sense Disambiguation Through Lattice Learning

by

Eli Stickgold

Submitted to the Department of Electrical Engineering and Computer Science
on February 1st, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

The question of how a computer reading a text can go from a word to its meaning is an open and difficult one. The WordNet[3] lexical database uses a system of nested supersets to allow programs to be specific as to what meaning of a word they are using, but a system that picks the correct meaning is still necessary. In an attempt to capture the human understanding of this problem and produce a system that can achieve this goal with minimal starting information, I created the DISAMBIGUATOR program. DISAMBIGUATOR uses Lattice Learning to capture the concept of *contexts*, which represent common situations that multiple words are found in, and uses Genesis' system of Things, Sequences, Derivative and Relations to understand some contexts as being related to others (i.e. that 'things which can fly to a tree' and 'things which can fly to Spain' are related in that they are both special cases of the context 'things which can fly'). Using this system, DISAMBIGUATOR can tell us which meaning of 'hawk' we should use if we see it in a sentence like 'the hawk flew to the tree.' DISAMBIGUATOR is implemented in Java as part of the Genesis system, and can disambiguate short stories of around ten related statements with only a single query to the user.

Thesis Supervisor: Patrick H. Winston
Title: Professor

Acknowledgments

Mom, Dad, Emma, Jessie, John, Debbie, Adam and Stephanie, for believing in me, and always putting up with me no matter how grumpy I got.

Patrick Winston, for always being full of helpful ideas, advice and wisdom.

My gaming groups, for being very tolerant of being targets of venting.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

- In this section, you will learn what portion of the problem of machine intelligence DISAMBIGUATOR is intended to solve. You will see a basic example of the kind of questions DISAMBIGUATOR solves, and get a non-technical explanation of the basic process it uses to arrive at its answers. You will also get an introduction to the terminology of DISAMBIGUATOR's components that will be used throughout this thesis.

- By the end of this section, you will have enough of an understanding of how DISAMBIGUATOR works to explain it to a non-technical audience. You will also have an outline of the rest of this thesis so that you know which sections to skip to if you are already familiar with the background material or only need the summary of my results.

1.1 Overview: Going from text to imagination.

Suppose I told you to think of a ship sailing out of a delta. Consider the thought that this conjures up in your mind. Most likely, you don't see the letters of the sentence, or recreate the sound that it would make, but rather you imagine the situation described.

Maybe you conjure up a picture or a video that you've seen before that this sentence describes, or maybe you've combined two memories to get it - you have some

memory of a boat sailing along a river, and you've seen a picture of the Mississippi delta, so you superimpose them in your imagination to produce a hallucination of the ship making the journey out of the Delta. It's possible you might imagine this in the form of a movie of sorts, a dynamic image, or maybe it's just a visual snapshot - if you have memories very close to the situation described, you may also hallucinate sounds or smells.

Figure 1-1: An image of a boat sailing may be superimposed over an image of a river delta to create a composite hallucination.

One thing you're unlikely to imagine is a boat sailing out of a giant Greek letter. You *could* probably imagine that, but it's a far less sensical interpretation of the text you've been given than your initial imagination. A computer, however, given just such a text, has no intrinsic way to convert the words it reads in into meanings. Without such an ability, the computer's ability to hallucinate from a text is severely reduced, because even in such a small sentence, there are a large number of possible combinations of word-meanings, each of which would conjure up a different image in a human imagination.

I believe that the ability to imagine with facility from nothing but text is a vital part of achieving true artificial intelligence. The ability to imagine is critical to human reasoning, as it permits us to manipulate scenarios, consider alternatives, predict outcomes and more without requiring experience of each possible situation considered. Our ability to transition from text to imagination provides a compelling case that it should be possible.

For small texts, such as those Genesis currently works on, it is possible for human workers to go over every text to be used and manually select a single meaning for each word from a list of possibilities, obviating the need for a program to do this. However, in the long run, in order to function, a system like Genesis will want to be

able to constantly process new texts from high-throughput sources like news articles, intelligence reports and blogs.

The amount of manpower it would take to perform manual disambiguation of this quantity of information would rival the amount necessary to produce it in the first place, making it infeasible to process by hand in the long run, so a computerized method of disambiguation is a necessity for programs intended to manage large streams of text-format information - a purpose that cannot be easily discarded, as most information humans produce for transmission is inevitably in text form, and it seems unlikely that we will change this fact.

1.2 Disambiguator simulates human imagination using meaning relations.

In the example of a ship sailing out of a delta, I mentioned that even if you had no memory of something explicitly detailing this precise occurrence, you might have some memory of a ship sailing out of some other feature of a landscape, making it relatively simple for you to replace that feature with another memory of a river delta. It would be harder, though not impossible, to imagine a ship sailing out of a greek letter, as that has much less in common with, say, a river or a harbor.

Harder still would be to imagine a ship sailing out of a change in value (another possible meaning of delta), because this meaning does not denote a tangible object, but rather an abstract quality. In a different statement, such as 'the hawk flew', it would be similarly difficult to imagine the use of hawk as a verb meaning to hunt, as it is difficult to picture an action flying.

DISAMBIGUATOR uses the WordNet[3] dictionary, which provides all the possible meanings of a word in the form of nested sets defining categories that the word can fall into depending on which meaning. The Genesis system uses these definitions to construct a bundle of *threads*[4] (see 2.1) to provide this measure of similarity. This similarity can be used to define an 'ease of imagination', or how natural it would be

for a human to mutate an existing memory into a hallucination of the desired scene. Using the method of lattice learning, DISAMBIGUATOR reinforces this measure with negative examples, allowing it to comprehend that some meanings might emphatically *not* make sense in certain contexts. In the example of 'the hawk flew', if the only known example of a thing flying we had was a bat, we might rightly say that the use of hawk meaning a militaristic politician was closer taxonomically to the use of bat meaning a mammal, but we know that most people we know of cannot fly, so we can discard that option in favor of the correct one.

In these ways, DISAMBIGUATOR makes meaning from unknown words by attempting to mimic our best understanding of how human minds deal with the same issue - by making use of existing knowledge both about other things seen in similar contexts and about how meanings relate to each other.

1.3 Disambiguator uses contexts and statements to track rules.

DISAMBIGUATOR uses two basic units of information to track its knowledge and use it to disambiguate words: the *context* and the *statement*. Throughout this paper I will use these terms as I describe DISAMBIGUATOR's operation, so I will explain their basic meaning here.

For my purposes a *context* defines a set of information describing - partially or completely - the words and syntactic structure surrounding a word. A context captures the idea that multiple words may occur in similar constructions, making it likely that their meanings should be somewhat related. The actual definition of a context is based on Genesis' system of representing sentence structure and will be explained more completely in chapter three, but in essence a context defines an attribute that a word may have, as simple as 'things that can fly' or 'things a ship can do', or as complicated as 'things that can land in a bush quickly'. Contexts are related in a heirarchical fashion by a subset/superset relation - so that 'things that can fly' is

understood to be a parent of 'things that can fly to a bush'. Over time, a context may be marked up with positive and negative examples, which will both serve to rule out some meanings within that context and also provide data for determining what the 'easiest to imagine' meaning of a word is.

A *statement* is a natural extension of a context, and specifies both a context and a set of possible meanings that must be considered in that context. A single sentence may expand into several statements - at least one for each word, possibly more if the sentence is complex - and a statement serves as the unit of disambiguation, providing a neatly packaged question that can be answered (or deferred) by DISAMBIGUATOR as it works through the story.

It is important to know that several statements can contain the same context, and that the context will evolve and learn from each of them - this is key to the design of the DISAMBIGUATOR program, as it is the root of its generalization capability.

1.4 What does Disambiguator disambiguate?

So what parts of a story will Disambiguator work on? In the simplest explanation, it will disambiguate any word that falls into a simple subject or verb position in a sentence. Its actual reach is slightly larger, and is defined by Genesis' representation of sentences (see 2.2).

Suppose we give DISAMBIGUATOR the following input:

- Assume 'fly' means 'move'.
- Know that a guitar cannot fly.
- The osprey flew to the tree.
- The bat flew to the bush.
- The hawk flew.
- The plane flew into the sky.

Pulling out only the words that fall into simple subject or verb positions, we can see that the following words will be disambiguated:

- Osprey
- Bat
- Hawk
- Plane
- Fly

The path descriptors - 'to the tree', etc. - on the other hand, will not be disambiguated. The reason for this is explained more fully in chapter four, From text to meaning.

1.5 What to read next: a map.

The rest of this thesis is divided into four sections. Here I give you a map of what to read next, depending on what you're looking for:

- Chapter two, Infrastructure, details the technology that DISAMBIGUATOR builds upon to achieve its goals, including the WordNet dictionary, thread terminology and Genesis' system of Things, Sequences, Relations and Derivatives. Read this section next if you are unfamiliar with these systems, as they are used heavily in the following sections of the text.

- Chapter three, Learning with maps, discusses the underlying systems of DISAMBIGUATOR and shows with examples how its structure captures the rules of imagination through positive and negative examples, and gives a demonstration of what the learning process looks like. You can skip to this section if you are familiar with the Genesis system and want to understand the underlying architecture of DISAMBIGUATOR

- Chapter four, From text to meaning, describes the actual implementation of DISAMBIGUATOR as part of the Genesis system and the specific rules it uses to turn the output of the system's parser into positive and negative examples from which to learn in the manner described in chapter three. Read this after chapter two if you are interested in the applications of DISAMBIGUATOR in the Genesis system, but it can be skipped if you are interested in DISAMBIGUATOR primarily for its basic theories on how to approach disambiguation.

- Chapter Five, Contributions, sums up the contributions I have made through the DISAMBIGUATOR code, both to the problem of word-sense disambiguation as a theoretical issue and to the Genesis project in specific. Skip directly to this chapter if you only care for a non-technical description of the workings of DISAMBIGUATOR.

Chapter 2

Infrastructure: The technology behind Disambiguator

- In this chapter, you will learn about the technology underlying the DISAMBIGUATOR program. Specifically, you will be introduced to the concept of lexical Threads as a means of encapsulating a possible meaning of a word and associated information about that meaning, and to Genesis' representations of syntactic structure as Things, Derivatives, Relations and Sequences.

- By the end of this chapter, you will be comfortable that you understand the base on which DISAMBIGUATOR is built, and that you know why these particular representations have been chosen for this program. You will be able to explain what the advantages of these representations are in a relatively technical fashion.

2.1 Defining meaning with threads.

The purpose of DISAMBIGUATOR is to make a translation from words in text into explicit and unique meaning. In order to do this, a method of defining meaning is required. Fortunately, Genesis has just such a method: Threads[4]. A thread is a heirarchical list of categories with each link representing an 'is-a' relationship from the child to the parent. In this way, a thread captures a single explicit meaning

associated with the last word in the list. Genesis uses the WordNet[3] dictionary to provide definitions and constructs threads out of them. Beyond the fact that Genesis already uses threads for tracking meaning, there are several reasons why threads are a good choice for representing meaning to DISAMBIGUATOR.

```
thing entity physical-entity object whole living-thing organism animal chordate vertebrate bird bird-of-prey hawk
```

Figure 2-1: Threads store data as a heirarchical list of sets, each related to its parent by the 'is-a' relationship. In this thread, a hawk 'is-a' bird.

2.1.1 Threads capture relationships between word meanings.

The structure of threads makes them ideal for representing the concept of similarity between word meanings. Threads make use of nested, increasingly broad categories that make it possible to determine how far we must go to find a common relation between two word meanings by looking how far up each thread's list of sets we have to look before we reach a set that contains both meanings. This mimics how the human mind might search for relationships between two meanings, saying 'X is like Y because both X and Y are examples of Z'.

All definitions in WordNet's database are rooted in two basic nodes 'thing' and 'action', which together encompass all the word meanings it has available. To construct a thread from a definition in WordNet, Genesis simply lists in order all the categories that the definition belongs to. Thanks to this, it is possible to view WordNet's collected information on meaning as a pair of trees, one rooted at 'thing' and one rooted at 'action', formed by joining threads together wherever they share an uninterrupted chain of identical nodes starting at the root. It is important, however, to realize the limitations of this view: nodes within these trees are labeled with English words, but these labels do not provide a unique identifier for the node - even in internal nodes. Because of the ambiguity of text in language (the very problem we are attempting to solve), multiple nodes may share the same name because the two (distinct) sets they define share a common name. The only unique identifier of a

node in this tree-structure is the chain that links it up to the root node of its tree.

2.1.2 Threads easily handle multiple word meanings.

Thanks to WordNet, it's simple to consider all the possible meanings of a word in a form that allows easy comparison to a reference or set of rules. From a word, Genesis can generate a *bundle*, a set of threads that defines the set of meanings that could possibly be ascribed to a word, by taking all the possible meanings produced by WordNet and constructing a thread for each. More formally, the bundle generated by a word in text is the set of all threads generated from WordNet's database that terminate at a leaf marked by that particular word.

```
thing entity physical-entity object whole living-thing organism animal chordate vertebrate bird bird-of-prey hawk
thing entity physical-entity object whole living-thing organism person adult militarist hawk
thing entity physical-entity object whole artifact sheet board mortarboard hawk
action act interact transact deal peddle hawk
action get capture hunt hawk
action exhaust expectorate cough clear-the-throat hawk
```

Figure 2-2: A bundle contains all the threads the terminate with a specified word. This bundle captures the fact that 'hawk' can mean a bird, a person, or any of a number of actions.

By creating a bundle (using Genesis' BundleGenerator class), we are able to retrieve every possible meaning for a given word in thread form, which neatly enables us to use our rules and existing 'memories' in a program to prioritize individual threads as being easier to imagine and rule out other threads as not possible meanings in the current context based on our understanding of it.

2.1.3 Threads are a plausible analogue of human memory.

The originators of the Thread Memory System, Vaina and Greenblatt[4], showed that threads are reasonable not only as a computationally useful tool for storing word information, but also as a model of how human cognition might seek to solve the same problem, making a thread-based system a good candidate for modeling human disambiguation. In specific, Vaina and Greenblatt demonstrated that damage

to threads caused problems for thread-based systems that were very similar to those shown in human patients with specific types of aphasia.

2.2 Defining structure in Genesis

DISAMBIGUATOR uses a stricter definition of context than simply proximity-based systems for word-sense disambiguation. By making use the parser already available through Genesis, DISAMBIGUATOR takes advantage of its system of frames to make use not only of the proximity of a word to others when considering how to identify its meaning, but also its syntactic position in the sentence surrounding it - invaluable for differentiating between, for example, the noun and verb meanings of a given word.

The parser output is rooted in four Java classes that together implement several useful syntactic frames.

- The Thing class defines a unique object with an associated bundle.
- The Derivative class denotes structure derived from a single subject by a relation defined by a second bundle. The subject of a Derivative may be any of these four classes.
- The Relation class also has a subject, and also has an object that can likewise be an instance of any of these four classes. The bundle of relation describes the relationship between the subject and object.
- The Sequence class is defined by an ordered list of elements, along with an identifier to allow the system to distinguish between different syntactic uses of a Sequence. The elements of a Sequence can be Things, Relations, Derivative, or even other Sequences.

2.2.1 Things use bundles to capture all possible meanings.

Things are the most basic unit in the parser's output structure. A Thing is approximately what it sounds like, a single object from a story wrapped up in a package

that contains our knowledge about it, but nothing about its position in a story or its relation to other things. Any object that appears in a story, whether or not it has tangible form, will be reified into a Thing for storage and further reference.

More technically, a Thing is defined by a unique name and a bundle containing all the threads that might apply to it. This bundle will include all the possible meanings that the root word being represented by the thing could take on, but it will also contain labeled threads denoting any features that the object may have - in this manner, a 'red dog' will be a single thing, with its bundle formed by all possible meanings of the word dog, plus the information that the dog in question has a feature 'red'.

2.2.2 Derivatives have structure and can describe places.

A Derivative, as you might expect, represents something derived from another object. For example, if we wanted to express the concept 'at the table', we could do so in a single Derivative with the bundle 'at' and the subject 'the table'. More generally, the subject of a Derivative can be a Thing, Relation, Sequence, or even another Derivative, as in the example shown in the figure below.

In the program, a derivative object is specified by three things: a unique name, a subject and a bundle defining the way the new concept is derived from the subject.

2.2.3 Relations expand the applications of Derivatives.

A Relation is similar in structure to a Derivative, but has an object as well as a subject. The bundle of a Relation describes the way in which the subject is related to the object. A Relation can be used to describe action - the sentence 'the hawk flew to the tree' can be seen as a Relation between 'hawk' and the path 'to the tree' (a Sequence, as we will see in the next section) defined by the bundle 'fly'. Relations can also describe relative moments in time. For example, 'The flight happened before the landing' could be seen as a relation between 'the flight' and 'the landing' describe by the bundle 'before'.

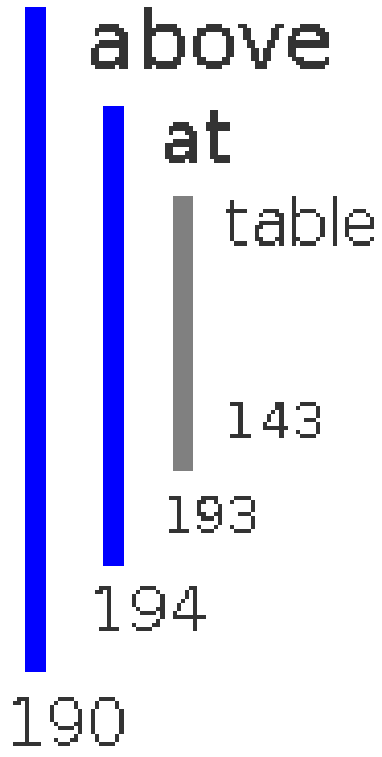

Figure 2-3: A derivative has a bundle describing the type of derivation and a subject, which can be a Thing, Relation, Sequence or another Derivative. In this example, The concept of 'above the table' is shown as a pair of nested Derivatives - 'above' the Derivative 'at the table'.

Since both the subject and object of a relation can be a member of any of the four classes listed, it is possible to use nested relations, derivatives and sequences to achieve extremely complex syntactic constructs without needing to resort to new classes. Due to its natural fit to the structure of an English sentence and its flexibility, Relations are used as the basis for contexts in DISAMBIGUATOR (see 4.2).

The bundle of a Relation also contains any features of the described relationship. This is important to us, as we make use of the fact that the Genesis parser turns a statement like 'the bird cannot fly' into a Relation where the bundle for 'fly' has the feature 'not'.

This construction is necessary to DISAMBIGUATOR as it lets the program accept negative examples in a simple fashion.

Figure 2-4: A Relation can accept an object as well as a subject. This can be used to represent a sentence like 'the hawk flew', as in this example, or to indicate time relations, such as 'the flight happened before the landing'.

2.2.4 Sequences allow representation of paths.

Finally, a *sequence* is just what it sounds like, an ordered list of other objects, which can be Things, Relations, Derivatives or other Sequences freely. Sequences contain less syntactic information in some ways than relations, as they don't specify the role of any element within the sequence, but they are invaluable in that they give the system a way to represent paths[1].

By using a sequence as the object of a relation we can express a concept such as 'the ship sailed via the delta into the ocean', which would otherwise be inexpressable, as it requires the complex understanding of a path as the object to the ship's motion, and a path cannot be easily represented without a construct such as a relation.

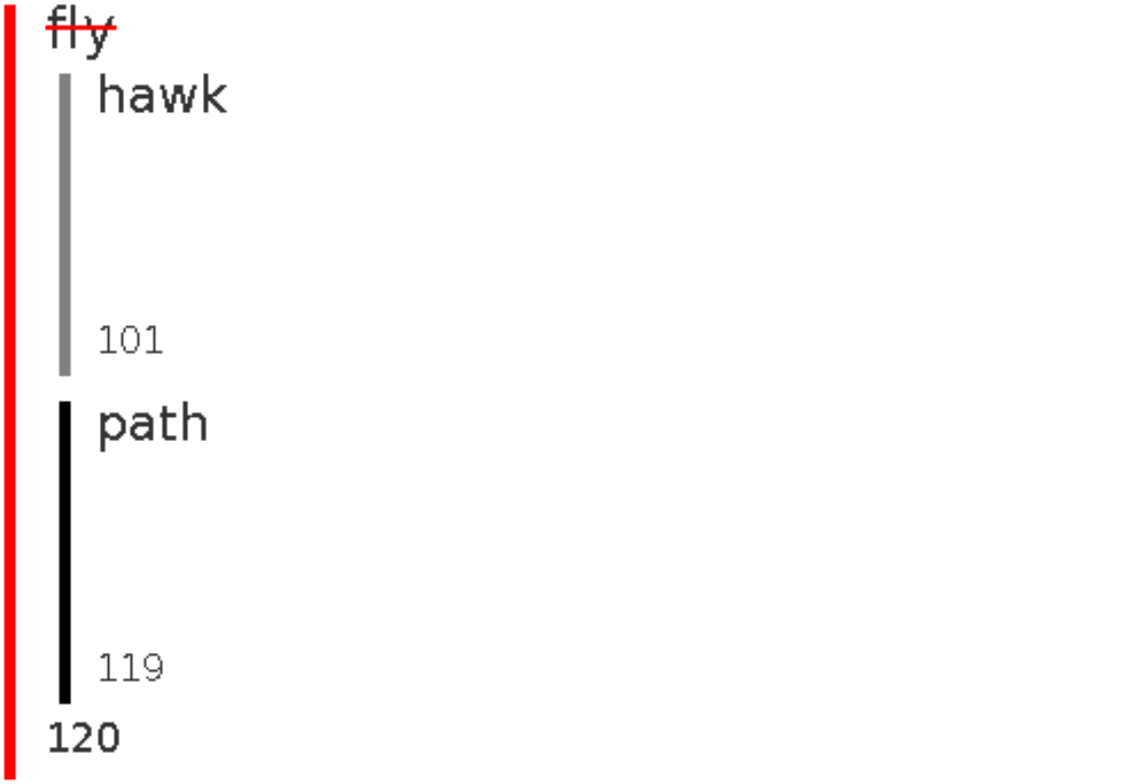

Figure 2-5: A bundle can contain 'features' that append information about it. Here, this Relation has the feature 'not', implying that it represents a Thing that cannot fly.

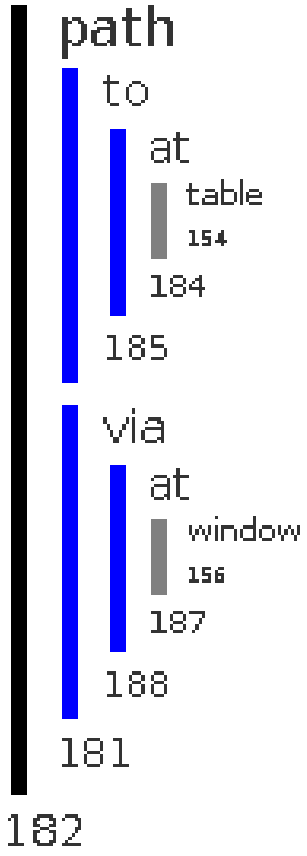

Figure 2-6: A complex path can be represented by a Sequence. Most commonly, a path will be found as the object of a Relation indicating movement.

Chapter 3

Learning With Maps

- In this chapter you will learn how DISAMBIGUATOR uses lattice learning on a system of concepts and statements to classify meanings and prioritize the meaning that makes the most 'sense' in the current context.
- By the end of this chapter, you will understand the concept of lattice learning and its uses for problems such as this, and be comfortable with the underlying concepts that make DISAMBIGUATOR work. You will be comfortable explaining the technical aspects of the DISAMBIGUATOR program to an informed audience and explaining the design choices that led to it.

3.1 Contexts specify situations in which meanings can occur.

As explained in the introduction of the thesis, a *context* in DISAMBIGUATOR is a pattern describing a class of sentences in which a word might be found. In a human sense, a context describes a type of memory into which we might attempt to insert a memory matching the meaning in question - in the example of a ship sailing out of a delta, the context of things which a ship sails out of can be thought of as the memory of a ship sailing, onto which we might attempt to project a separate memory of a view of a river delta.

DISAMBIGUATOR's implementation of contexts draws on the frames passed to it after the Genesis program has parsed a section of human text, drawing on the fact that a series of sentences in English will be represented as a list of Relations by the parser, roughly containing the role of subject and object in their appropriate slots, with the verb as the bundle assigned to the Derivative itself. DISAMBIGUATE takes on the problem of disambiguating words that appear within Things, Derivatives and Relations, and so concerns itself with contexts relating to these structures.

A Thing, by its nature, contains no context beyond the word and features contained in its bundle. A Derivative or Relation, on the other hand, gives a structure and a set of other words surrounding a given word, which can provide insight into its meaning. As the parser interprets sentences as Relations, we construct contexts out of Relations.

A context consists of two components: a rule defining when a context can be matched to a word from the text, and a set of information for choosing which meaning can be most easily imagined as occurring in that particular context.

The first rule is key to matching contexts to words in order to form statements in a regular fashion, so that information learned about a context from one word can be applied to another word later in the text that occurs in the same context. Each context is identified by two pieces of information. The first identifies which position in the relation is the one that contains the word that generated the context: 'SUBJECT', 'OBJECT' or 'VERB' (where 'VERB' refers to the bundle of a Relation itself). For example, the context 'things that can fly' would have the identifier 'SUBJECT', because it defines a property of the subject of the sentence. The second identifies a set of restrictions on what sentences can match the context: either or both of the positions not identified in the first piece of information can contain a Thing, Derivative, Relation, Sequence or bundle (in the case of the verb position), and the context will only match with sentences that have the same object in that position. To continue the example, 'things that can fly' is defined by the fact that it is referring to words in the subject position, and that it can only match sentences whose Relations have the bundle 'fly'. The context 'things that can fly to a tree' would likewise have

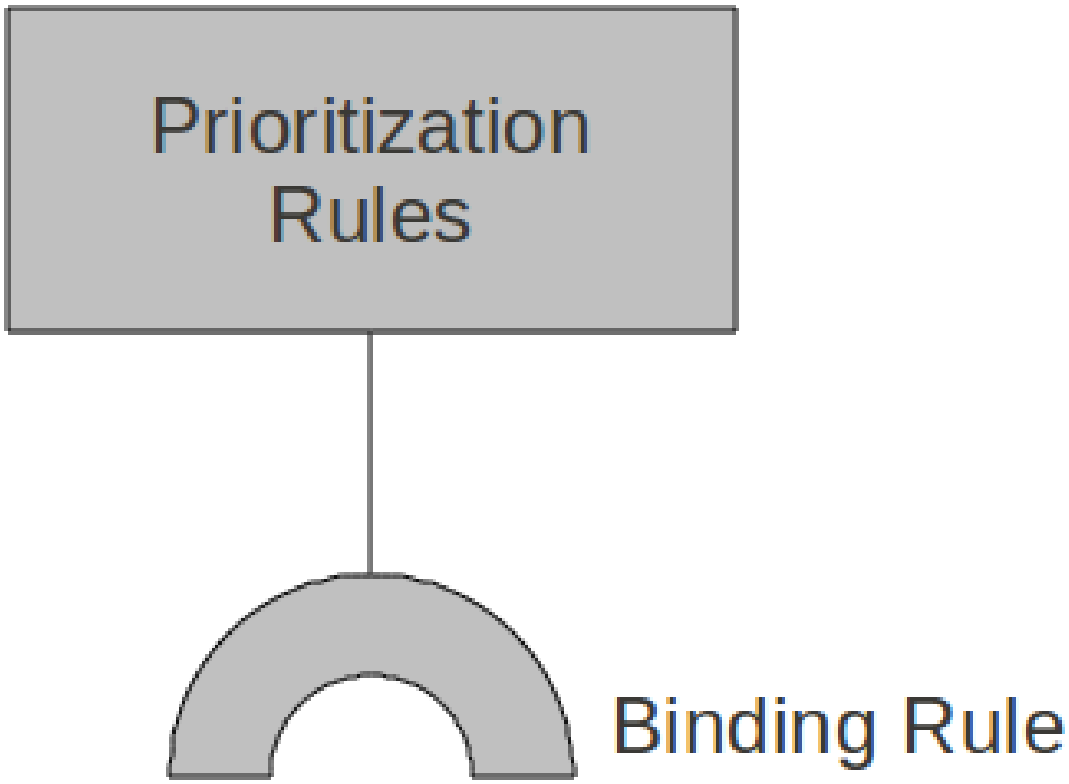

Figure 3-1: A context consists of a rule for whether or not the context will bind to a word in text, and then a series of tests for picking a meaning for the word.

the subject identifier, but it would only match to sentences whose Relations have the bundle 'fly' and the object 'to the tree' (a Sequence defining a path).

In this way, a single instance of a word in the text may fit into several possible contexts. For example, in the sentence 'the hawk flew to the tree', the word 'hawk' fits not only into the context of things that can fly to trees, but also into the context of things that can fly and things that can take some sort of action to a tree. All three of these contexts will be related in the graph of contexts formed by DISAMBIGUATOR (see 3.4).

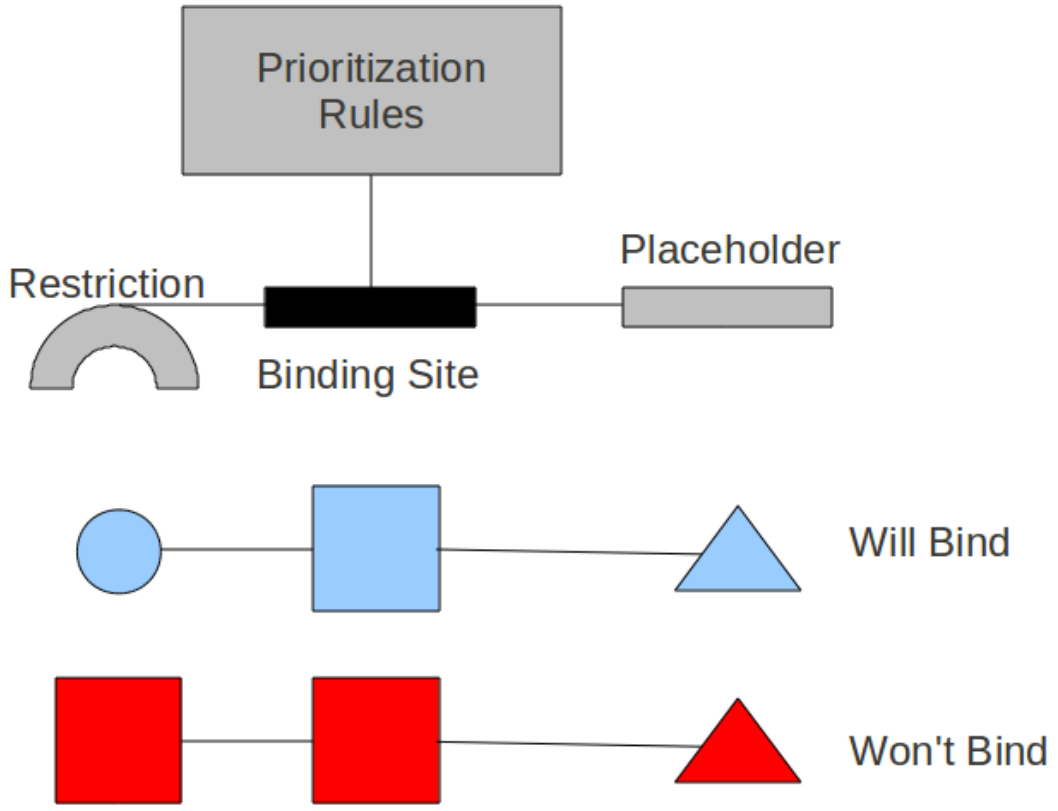

Figure 3-2: A context has a specific place in which a word may bind, and one or two restrictions on the words or Things that can surround the specified word.

3.2 Lattice learning creates categories from examples.

When a context is matched with a word in a statement, its second piece comes into play with the purpose of determining which meaning for the word should be selected. This takes the form of two different methods, which are applied in turn. First, the context determines which, if any, of the possible meanings can be imagined at all in the context specified - that is to say, which meanings are not ruled out by previous negative examples. Second, if there is ambiguity remaining, the context determines how far removed from the closest positive example each possible meaning is, which provides a basic metric of how 'hard to imagine' that meaning is.

In order to learn at some basic level what meanings make sense in what context, a computer program wants to be able to associate a context with a boolean function that specifies whether it is able to 'imagine' a specific meaning withing a certain concept. For example, within the context of 'things that can fly', we would like a boolean function fly, such that

$$fly(bird) = True$$

$$fly(person) = False$$

$$fly(action) = False$$

In order to create this function, DISAMBIGUATOR uses a system of classification called lattice learning to create decisive functions from relatively little information. The lattice learning algorithm[2], updated slightly for this thesis to provide the ability to recognize the concept of exceptions to a rule. The concept of Lattice Learning is very simple. Treating the theoretical assembly of all possible threads as a pair of trees, it classifies nodes into three possible categories: nodes about which nothing is known yet, nodes that contain among their eventual leaves a negative example, and nodes that contain among their eventual leaves only positive examples.

Thanks to the fact that there are only two possible trees in WordNet, Lattice Learning requires very few examples before it can classify any given thread into either a positive or negative category.

Lattice Learning is also valuable in that it makes very good use of carefully chosen examples, magnifying the effectiveness of any knowledge laid down by a human before the story through use of the concept of near-miss learning[5]. For example, when trying to nail down the concept that birds are a category of creatures that can fly, lattice learning requires only two examples to narrow down to the proper node.

One primary weakness of the lattice learning algorithm is that because it gives a negative status to a node that contains *any* negative leaves, it proves vulnerable to error when dealing with categories that are not strictly nodes in WordNet. For

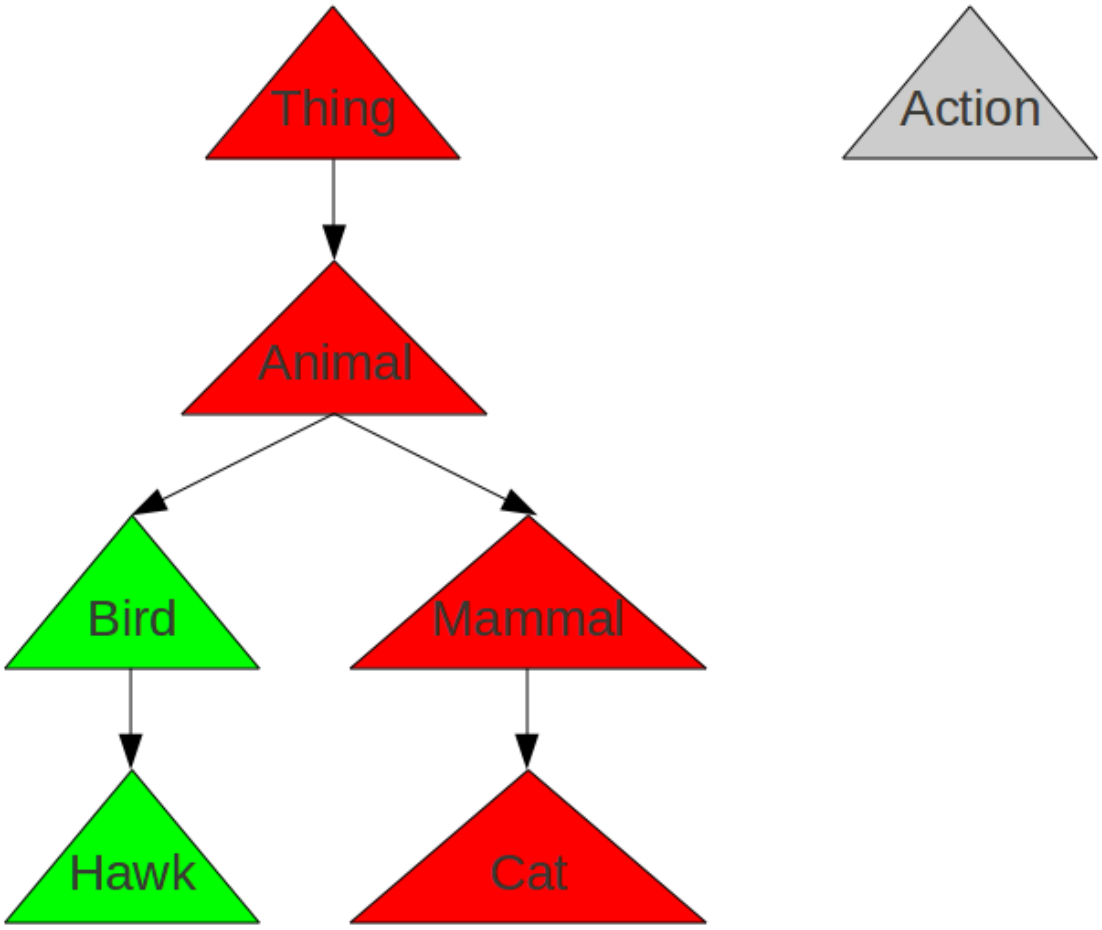

Figure 3-3: Lattice learning classifies nodes into unknown, positive or negative. In this example, the action tree is fully unknown. The node 'bird' is positive because it contains the positive example 'hawk' and no negative examples, but the node 'animal' is negative because it contains the negative example 'cat'.

example, when trying to isolate things that can fly, if the algorithm receives 'ostrich' as a negative example, it will immediately rule out 'bird' as a possible generic category of things that can fly. Once ostrich is given as an example, the algorithm will reject any birds it is asked about except those that have specifically been labeled as positive examples.

Fortunately, it is possible to correct this issue relatively simply. Rather than strictly ruling out any node that contains a negative weight, nodes are assigned an integer value, rating how much the algorithm believes that it does or does not match the required quality. Negative examples apply a negative modifier to the scores of

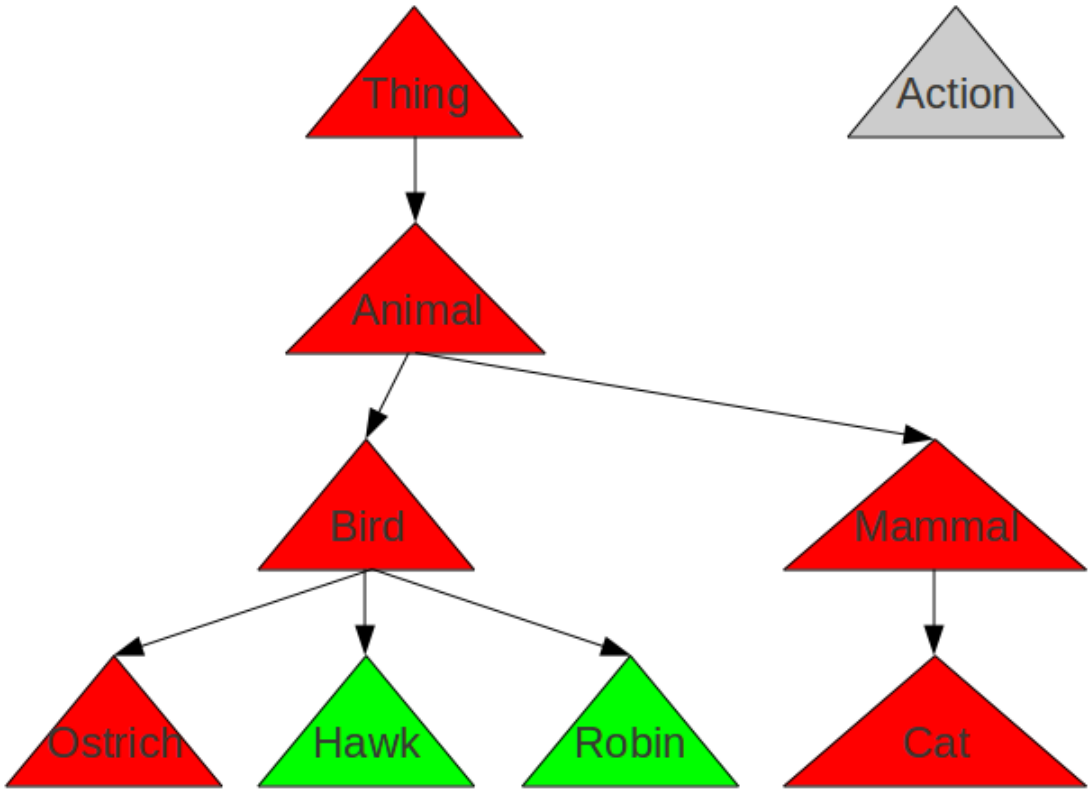

Figure 3-4: Once ostrich has been given as a negative example, by default bird will never be a positive node no matter how many positive examples we see.

all nodes that contain them (found by simply walking 'up' the thread), and positive examples apply a positive modifier. In order to maintain an algorithm that mostly functions in the same fashion as lattice learning, negative values are assigned a higher weight, meaning that a node that contains negative leaves will only have a non-negative score if it has a much higher preponderance of positive leaves.

3.3 Ease of imagination provides a second layer of disambiguation.

As a context receives positive and negative examples, it generates a lattice learning network from them, which it then uses to classify further meanings as either possible (positive scores) or not (negative scores). However, this distinction is not always

sufficient to narrow down a bundle of possible meanings to one possible one. Likewise, in the human mind, it is not always the case that it will prove *impossible* to imagine any of a number of possible meanings for a word, but it will likely prove more easy to imagine one meaning than another.

To provide a second layer of determination, a context also keeps track of all positive examples it receives over the course of DISAMBIGUATOR's operation. If DISAMBIGUATOR is unable to make progress using only the strict boolean functions contained in its contexts, it will fall back on a means of approximating this measure of 'ease of imagination' by determining how far up the nearest positive example we have to go to find a set big enough to contain the meaning being tested. While this method may not be as strict in selecting the correct meaning as the lattice learning rules, it helps enable DISAMBIGUATOR to better mimic human behavior in swiftly selecting a given meaning to imagine even in ambiguous surroundings.

3.4 Contexts are related in a map structure.

Although each context maintains its own set of examples and its own function derived from the lattice learning algorithm, an individual context does not exist in a vacuum. Just as part of the strength of the context system is that it makes use of our knowledge of how words are related, we improve the breadth of knowledge available to individual contexts by making use of knowledge as to how contexts are related to each other.

By their nature, contexts are formed from bundles, rather than meanings, so we cannot directly exploit the thread system to relate similar contexts to each other, but we can make use of the structure available from the parser to related contexts that define subsets or supersets of each other. To do this, we consider the set of all contexts as nodes in a directed graph. We define a parent-child relationship on the map of contexts as follows: an edge goes from one context to another if the second context can be considered a direct subcase of the first context. In our Relation-based implementation of contexts, we define the parent-child relationship by looking at the rules that determine what words can bind to the given contexts. In formal definition,

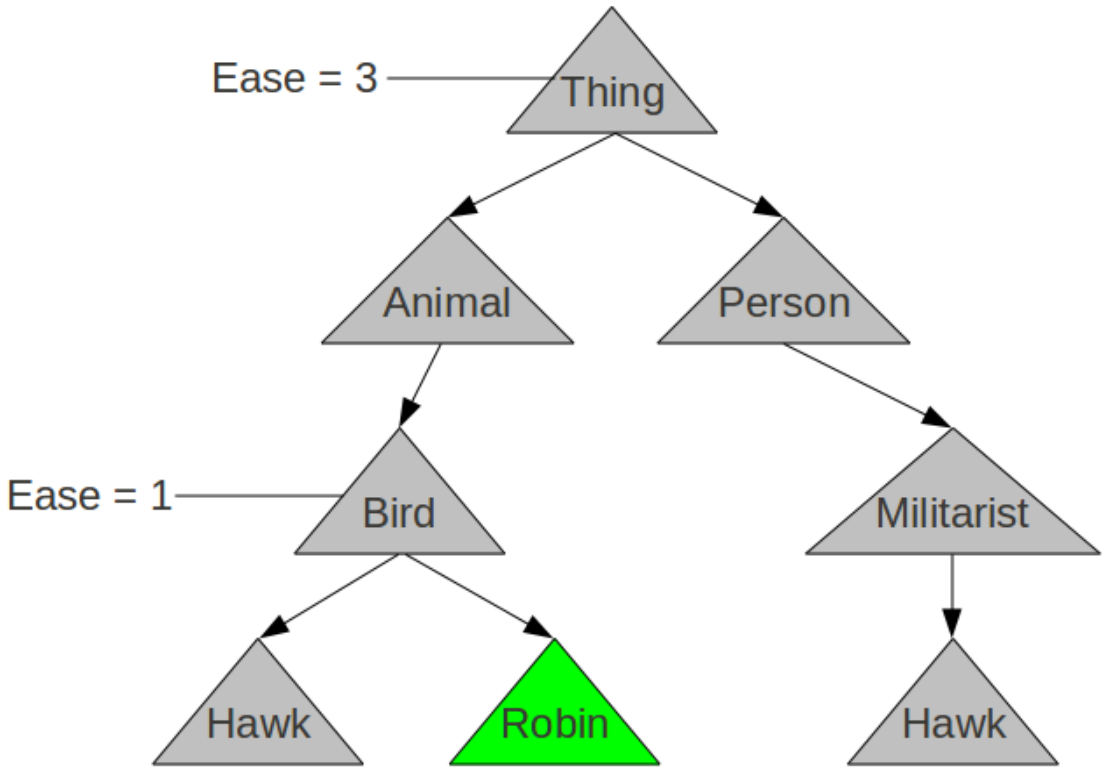

Figure 3-5: 'Ease of imagination' captures how far up the nearest positive example a context has to look before it finds a set large enough to contain the meaning in question.

if any word that can bind to context A can also bind to context B, then A is considered a child of B and B is likewise considered a parent of A. This is implemented by the simple expedient that a context is considered a child of another context if its binding rule can be reached by taking the other context's binding rule and filling it one of its placeholders with a specific Thing.

By our definition of contexts, the result of these connections between contexts is that any positive example given to a context propagates 'upwards' through the map of contexts, going against edge direction (such that any node which has a path to the affected context will be affected) and any negative example will propagate 'down' the map, such that any node that can be reached from the starting context is affected.

This structure mimics our human understanding of what a context means. If something falls into a specific category defined by a context, then it definitely falls

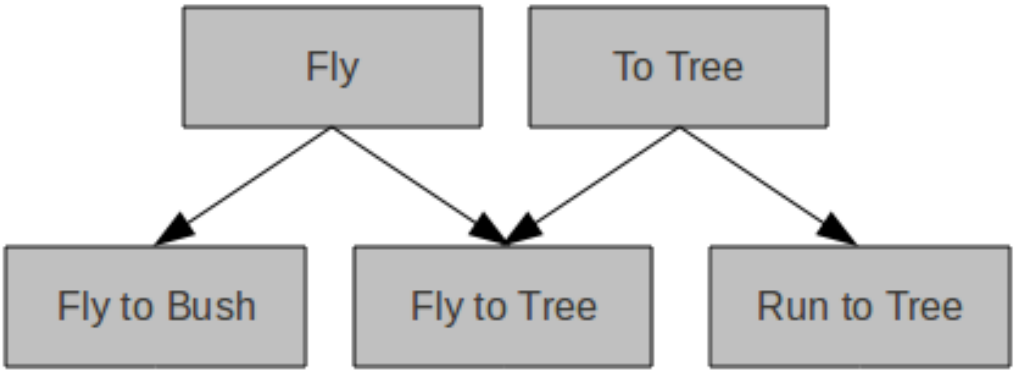

Figure 3-6: Positive examples propagate upwards through the net of contexts, while negative examples propagate downwards through the net.

into any general category that surrounds that context, so if a bird-hawk is a thing that can fly to a tree, it is definitely also a thing that can fly. Similarly, if something does not fit into a general category, it will not fit into any sub-category that can be derived from it, so if a person cannot fly, then a person cannot fly to a tree.

Chapter 4

From Text to Meaning

- In this chapter, you will learn how DISAMBIGUATOR makes the transition from human-readable text to assignments of meaning. You will see how DISAMBIGUATOR constructs the contexts described in 3.1 from the output of Genesis' parser.

- By the end of this chapter, you will understand the full path taken from human-readable story to disambiguation key. You will be able to explain the full workings of DISAMBIGUATOR to a technical audience.

4.1 Genesis parses text into Things.

The first step in the process of disambiguation is not performed by DISAMBIGUATOR at all, but rather by the Genesis program and its parser. Genesis operates on a modular system, and so when a story is loaded into it, it is first 'read' by the parser, which translates the entire story into an instance of the Sequence class, containing all the information contained from the statement 'begin story' to the statement 'the end'. The parser itself has no ability to disambiguate words into meanings, so all words in the text are stored as bundles rather than unique threads, containing all possible meanings.

The interface between Genesis and DISAMBIGUATOR takes the Sequence produced

by the story and converts it to a list of Derivatives (some of which may be Relations). It is this list of Derivatives that is passed to DISAMBIGUATOR itself.

4.2 Disambiguator creates contexts and statements from Derivatives.

DISAMBIGUATOR recieves the list of Derivatives and uses it to exhaustively construct a set of all possible non-trivial contexts (which is to say, all contexts that are not entirely made of placeholders) which can match any word in the text. This set of contexts is matched up with the words in the text to produce an exhaustive list of statements consisting of all possible word/context bindings. This list of statements is the list upon which DISAMBIGUATOR operates.

Each statement is also tagged with a piece of information derived from the features of the verb associated with it - specifically, it tracks whether the statement represents positive or negative information about the relation of the word in question to the context. In this way, the statement system is able to track a sentence like 'a person cannot fly' while realizing that it pertains to the same context as the sentence 'a hawk flew'.

4.2.1 'Assume' syntax disambiguates words in advance.

The Genesis parser accepts a specific syntax of the form 'Assume X means Y' that informs the parser of how to interpret the meaning of the word 'X' throughout the story. 'Y' must be a word that uniquely occurs in only one of the threads that make up all the possible meanings of 'X', and the result is that the parser restricts the bundles associate with 'X' to only the thread specified by 'Y'. For example, if a story began with 'Assume 'Hawk' means 'Bird',' then throughout the story, the Things produced based on 'Hawk' would have only the bird-specific thread in their bundle.

4.2.2 'Know' syntax forces early evaluation.

In order to allow DISAMBIGUATOR to accept prior knowledge about contexts, should we wish to give it such, the program accepts a special sort of syntax which is recognized after the parser has functioned, rather than before. Specifically, any sentence of the form 'Know that X', where X will be a Derivative itself, is treated as a piece of information to be acted on before the bulk of disambiguation commences. As such, before any of the rest of the story is processed into contexts and statements, sentences of this format will be removed, processed into contexts, and turned into positive and negative examples to seed the appropriate contexts with information.

4.3 Meanings are disambiguated in four phases.

Once DISAMBIGUATOR has its list of statements, it proceeds in cycles through the list. In each cycle, it looks over the full story and teststests for four possible ways to disambiguate words in order of surety. Lower-priority methods will not be used until no higher-priority methods produce any progress at any point in the story, so as to ensure the most accurate possible disambiguation.

4.3.1 Phase one learns from unambiguous words.

Phase one is the simplest of the possible steps, and contains two rules, either of which may be applied without favor. The first is simply the check to see if a statement has only one possible meaning - not all words produce more than one thread, so disambiguating them is trivial, but not unimportant, as they then provide examples for all contexts they appear in. Similarly, the second checks in the log of already disambiguated words to see if the word already has been assigned a meaning - this makes the implicit assumption that in a given story, each word will have only one possible meaning. Either of these rules can be applied without making a choice and without ambiguity, so they are the first rules to be applied.

4.3.2 Phase two uses the rules of lattice learning.

Phase two uses the boolean function created for each context by the lattice learning algorithm (see 3.2). For this phase, each statement checks all its possible meanings with the function associated with its context. If only one possible meaning is not considered non-sensical, then the sole sensical meaning is picked as the correct disambiguation for that word, recorded, and added to the appropriate context as an example. If more than one meaning in a statement is sensical, or if none are, phase two passes over it and move on to the next phase.

4.3.3 Phase three uses ease of imagination.

Phase three serves to disambiguate cases that passed by phase two by having more than one possible sensical meaning. When multiple meanings could be possible, phase three uses the metric of 'ease of imagination' (see 1.2) to choose between the possible meanings and select the one closest to a known positive example. That meaning is then recorded as the proper meaning to associate with the word and added as an example to the associated context.

Phase three will fail to disambiguate a statement if it has multiple meanings each the same distance from a positive example, or if it has no sensical meanings (distance is not considered if the meaning is rejected by the lattice learning function from phase 2).

4.3.4 Phase four requires user input.

Phase four is only activated if no statements can be disambiguated by any of the previous three phases - which is to say, when Disambiguator is stuck. In phase four, the system prompts the use to select a meaning from a dropdown menu of all possible meanings for a word in a provided context. If phase four is disabled, then Disambiguator will simply return only a partial mapping of words to meanings. So long as it is left active, however, Disambiguator will always eventually return a full disambiguation of the story in question.

4.4 Disambiguator in action.

Putting all the pieces together, I show here a full example of how the DISAMBIGUATOR program functions when given a small story - taken from a children's book about flying things and modified slightly so that it is forced to use all the phases described above.

Suppose we give DISAMBIGUATOR the following input:

- Assume 'fly' means 'move'.
- Know that a guitar cannot fly.
- The osprey flew to the tree.
- The bat flew to the bush.
- The hawk flew.
- The plane flew into the sky.

Pulling out words that fall into Derivatives and Relations, want to disambiguate the following words:

- Osprey
- Bat
- Hawk
- Plane
- Fly

4.5 First step: 'Assume' disambiguations.

We have already given DISAMBIGUATOR one disambiguation to start it on its way - we have told it that it should assume that we want the meaning of 'fly' that contains 'move' (rather than, say, the bug). The parser processes this before DISAMBIGUATOR ever even sees the story, so the word 'fly' will already be unambiguous.

4.6 Second step: 'Know' is processed first.

Because we have a sentence with the 'know that' key phrase at the start, it is processed first. Guitar is an unambiguous word, so the program accepts it as a negative example for the context 'things that can fly'. It also accepts 'fly' as a negative example of 'things a guitar can do', but as the word guitar never shows up again, this context is never used.

4.7 Third step: Main disambiguation.

Now the program enters its main disambiguation loop. Because of the careful design of this example, all four possible rubrics for choosing a meaning are used over the course of the run.

4.7.1 Phase one: Unambiguous words.

Both 'osprey' and 'fly' have only one meaning in their bundle ('osprey' is naturally unambiguous and 'fly' has been already disambiguated by the parser thanks to our 'assume' command), so in the first two passes, these words are assigned the only possible meanings and contexts surrounding them are updated. The important context that is updated that we will use later is, once again, 'things that can fly', which now has one positive and one negative example.

4.7.2 Phase two: Lattice learning.

Next up is 'bat'. 'Bat' has several possible meanings (the action in baseball, the club, or the flying mammal), but only one of them is judged possible by the lattice learning function attached to the context 'things that can fly'. The negative example of 'guitar' rules out the option containing the node 'artifact' (the club), and we have no information about actions, so we select the option containing the node 'animal'.

4.7.3 Phase three: Ease of imagination.

After ‘bat’, ‘hawk’ is disambiguated. Two meanings of ‘hawk’ make it past the lattice learning filter: the bird, and the militarist. However, since we have the positive examples of ‘osprey’ and ‘bat’ now, the program rules that the bird meaning is more likely, as the node ‘bird-of-prey’, where ‘osprey’ and ‘hawk’ intersect, is lower than the node ‘living-thing’, where ‘osprey’, ‘bat’ and ‘hawk’ intersect under the other definition.

4.7.4 Phase four: user query.

This leaves only one word remaining: ‘plane’. But unfortunately, our lattice learning program has by now ruled out all man-made artifacts as not capable of flight. With no answer available, the program uses its last recourse, and prompts the user to provide the proper selection from a list.

Chapter 5

Contributions

5.1 Results

In a final test, I demonstrated that DISAMBIGUATOR could take a short story designed to require every section to function properly and would disambiguate all target words. The full text of this example can be found in the first appendix of this thesis, with a step-by-step analysis of DISAMBIGUATOR's operation.

More typical (but less exhaustive) testing has shown that DISAMBIGUATOR can disambiguate a short story of 5-10 sentences on a coherent topic entirely with only a single query to the user.

5.2 Contributions

In this section I list the contributions made in my thesis:

- Expanded the Lattice Learning algorithm to be more robust in handling negative exceptions to positive rules.
- Proposed the effectiveness of Lattice Learning as a technique for word sense disambiguation with low initial information.
- Implemented DISAMBIGUATOR as a component to Genesis in Java.

- Demonstrated the effectiveness of DISAMBIGUATOR at disambiguating words in a story with a coherent theme.

Bibliography

[1] Ray Jackendoff. *Semantics and Cognition*, chapter 9. MIT Press, 1983.

[2] Michael T. Klein Jr. Understanding english with lattice learning. Master's thesis, Massachusetts Institute of Technology, 2008.

[3] George A. Miller. Wordnet 3.0. www.wordnet.princeton.edu, 2006.

[4] Lucia Vaina and Richard Greenblatt. The use of thread memory in amnesic aphasia and concept learning. *MIT Artificial Intelligence Laboratory*, 1979.

[5] Patrick H. Winston. *Learning Structural Descriptions From Examples*. PhD thesis, Massachusetts Institute of Technology, 1970.