Visualizing Inference

by Joseph D. Henke

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 2014
[June 2014]

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
May 15, 2014

Signature redacted

Certified by .....
Henry A. Lieberman, Principle Research Scientist, Thesis Supervisor
May 15, 2014

Signature redacted

Accepted by .....
Prof. Albert R. Meyer, Chairman, Masters of Engineering Thesis Committee

Visualizing Inference

by Joseph D. Henke

Submitted to the Department of Electrical Engineering and Computer Science

May 15, 2014

In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Common Sense Inference is an increasingly attractive technique to make computer interfaces more in touch with how human users think. However, the results of the inference process are often hard to interpret and evaluate. Visualization has been successful in many other fields of science, but to date it has not been used much for visualizing the results of inference. This thesis presents Alar, an interface which allows dynamic exploration of the results of the inference process. It enables users to detect errors in the input data and fine tune how liberal or conservative the inference should be. It accomplishes this through novel extensions to the AnalogySpace framework for inference and visualizing concepts and even assertions as nodes in a graph, clustered by their semantic relatedness. A usability study was performed and the results show users were able to successfully use Alar to determine the cause of an incorrect inference.

Acknowledgments

Firstly, I'd like to thank all the researchers and users who contributed to Conceptnet as part of the Media Lab's Common Sense Computing Initiative. Their work in aggregating the data and making it accessible was immensely helpful in my research.

Next I'd like to thank the researches who developed AnalogySpace. This work is entirely grounded in the AnalogySpace's framework for inference, and so without it, this thesis would not have been possible. In particular I'd like to thank Rob Speer for his very illustrative Divisi2 code and documentation.

I of course would like to thank my parents. Their understanding and supportiveness have been the foundation which has made everything I've done at MIT possible. Mom, I am still learning so many life skills from you. Thank you for always being there when I need help. Dad, from the mathematical sanity checks to helping me "make decisions right," I couldn't have done it without you. Thank you both so much.

Lastly, I'd like to thank my advisor, Henry Lieberman. His fantastic knowledge and experience in the research word was invaluable in my work. Henry, I appreciate you always working with my big ideas and helping me make them into something more realistic and often better. You always pushed me in the right direction for next steps in my project. I especially thank you for working with my unique time constraints and enabling me to still produce something worthwhile. I wish you the best of luck in your continued research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To see how common sense reasoning could be leveraged using the results of this thesis, consider the motivating example of Sam, who runs a blogging website and would like to be able to suggest related blogs to users. There are many relatedness metrics Sam could use, however Sam would like to be able to suggest blogs which are *semantically* similar as well as fine tune the parameters of the process. Additionally, Sam would like to be able to determine why two blogs were incorrectly found to be related and correct the problem. This section illustrates how Sam could use Alar, this thesis's tool for dynamically exploring the results of common sense reasoning, to accomplish these goals by comparing different concepts and statements found in each blog.

1.1 Using Alar

This section illustrates typical workflows Sam may go through while using Alar. A example image of Alar can be found in Figure 1-1.

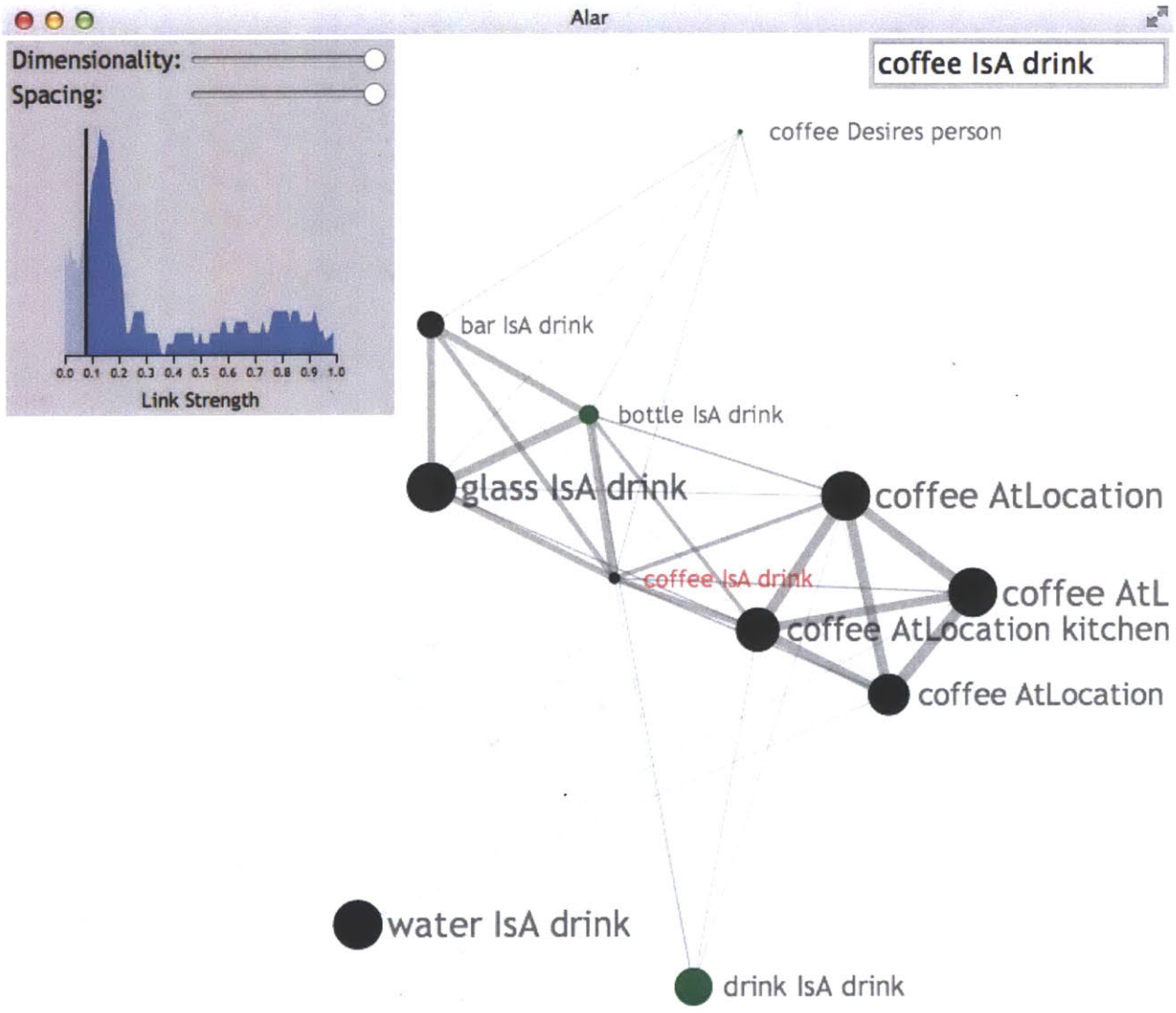

Figure 1-1: Alar's interface.

1.1.1 Exploring Concepts

One of Sam's popular blogs is about different coffee shops, so Sam would want to explore what concepts the inference process might find to be related to coffee. Sam would start by searching for "coffee," using the search bar in the top right of the interface. The interface autocomplete's Sam's partial query, indicating which concepts it is aware of as shown in Figure 1-2.

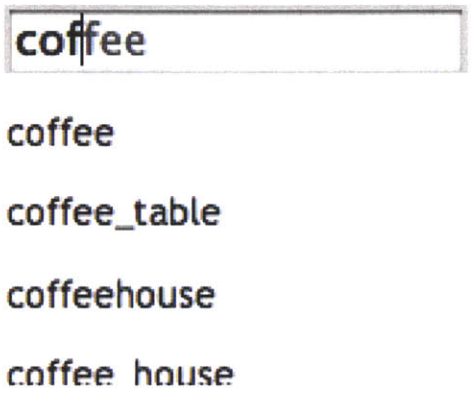

Figure 1-2: Alar autocomplete's partial queries, indicating which concepts it is aware of.

This will add "coffee" to the center of the graph as well as a handful of related concepts, as seen in Figure 1-3. To see what concepts were inferred to be similar to other concepts, Sam would look at the clustering of the nodes, as Alar clusters related concepts close to each other. This clustering is determined by the links, which pull nodes together, however not many links are being shown, so not much information about the relatedness of these concepts is being revealed. This can be seen in both the graph itself as well as the link distribution in the top left. This distribution shows all the possible links which could be displayed and their strengths. To avoid showing too many links and cluttering the interface, Alar has a threshold which indicates how strong a link must be, which is to say how related two concepts must be, to merit showing a link.

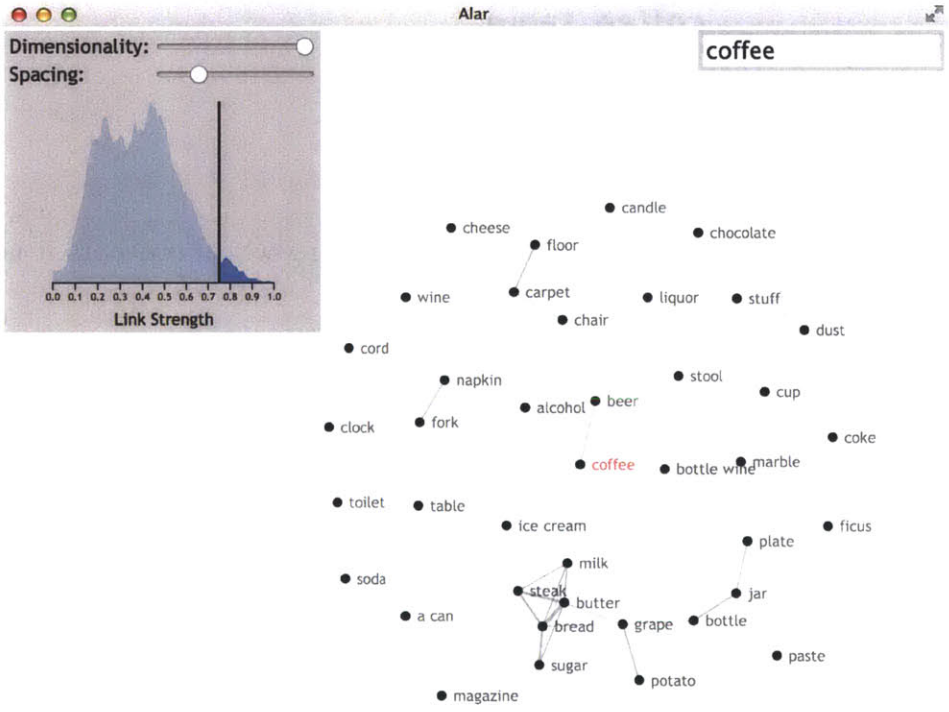

Figure 1-3: The interface after searching for "coffee."

To make Alar reveal more information about the results of the inference process, Sam would do the following. First, to allow more links to be shown, the link strength threshold could lowered, enabling more effective clustering of the nodes. However, because there are more links, the nodes are pulled tightly together, so to compensate, Sam can increase the spacing of the nodes using the spacing slider in the top left of the interface. This does not change anything about the links, rather it simply introduces more space between nodes. The results of these adjustments can be seen in Figure 1-4.

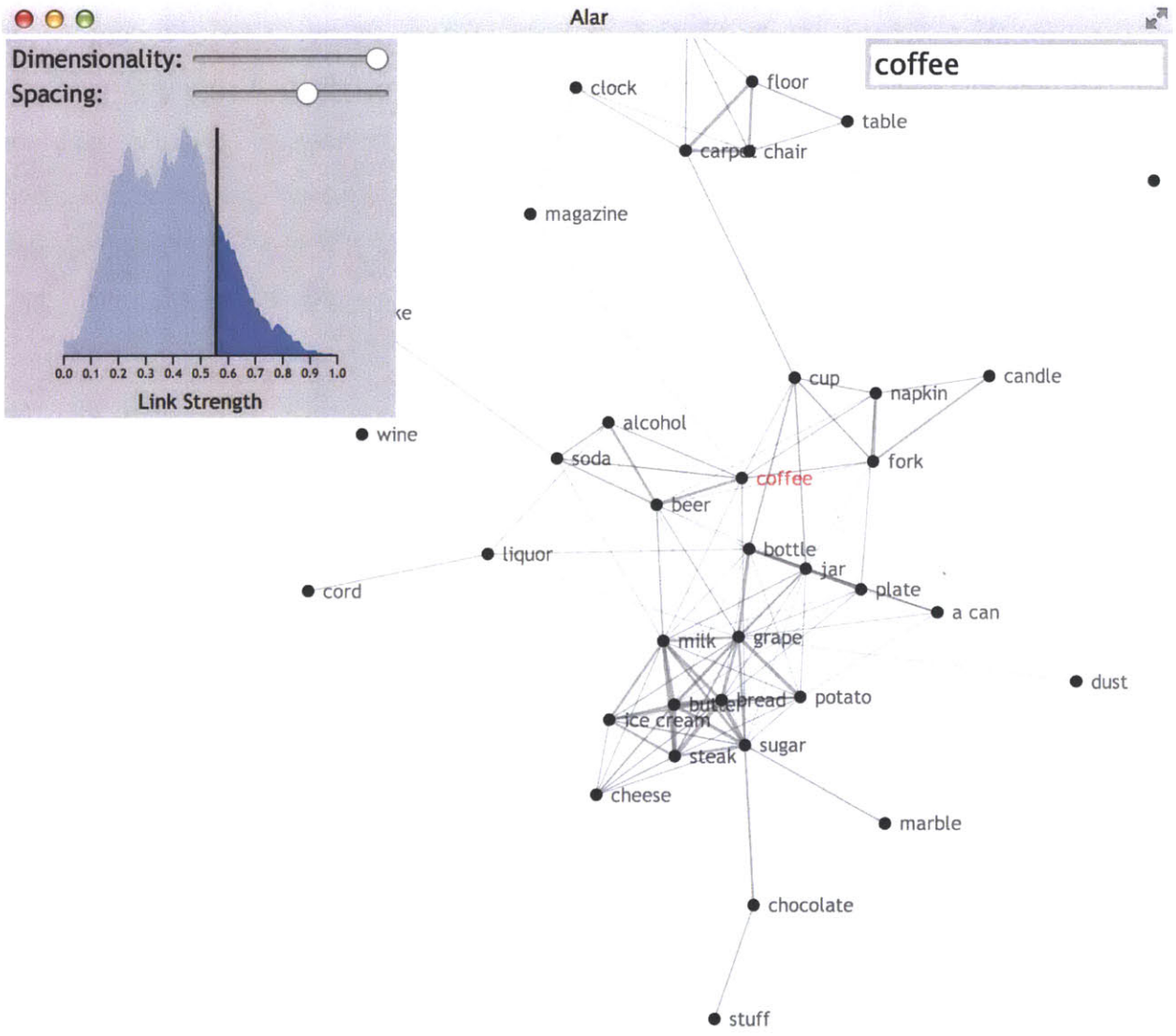

Figure 1-4: The interface after including more links and increasing the spacing provides much more information.

Now, Sam can more clearly see the concepts which are most related to coffee as well as the relationships of all concepts currently displayed. For example, "jar," "plate" and "can" are shown as being very close, which makes sense and gives Sam validation of successful inference in that area.

Next, Sam would like to go from just exploring concepts to exploring assertions.

1.1.2 Exploring Assertions

Moving from concepts to assertions, this workflow shows how Sam could explore assertions that were found to be related to "coffee IsA drink."

Much as in the previous workflow, Sam would simply search for the assertion and adjust the visualization parameters to make it more readable, as in Figure 1-5.

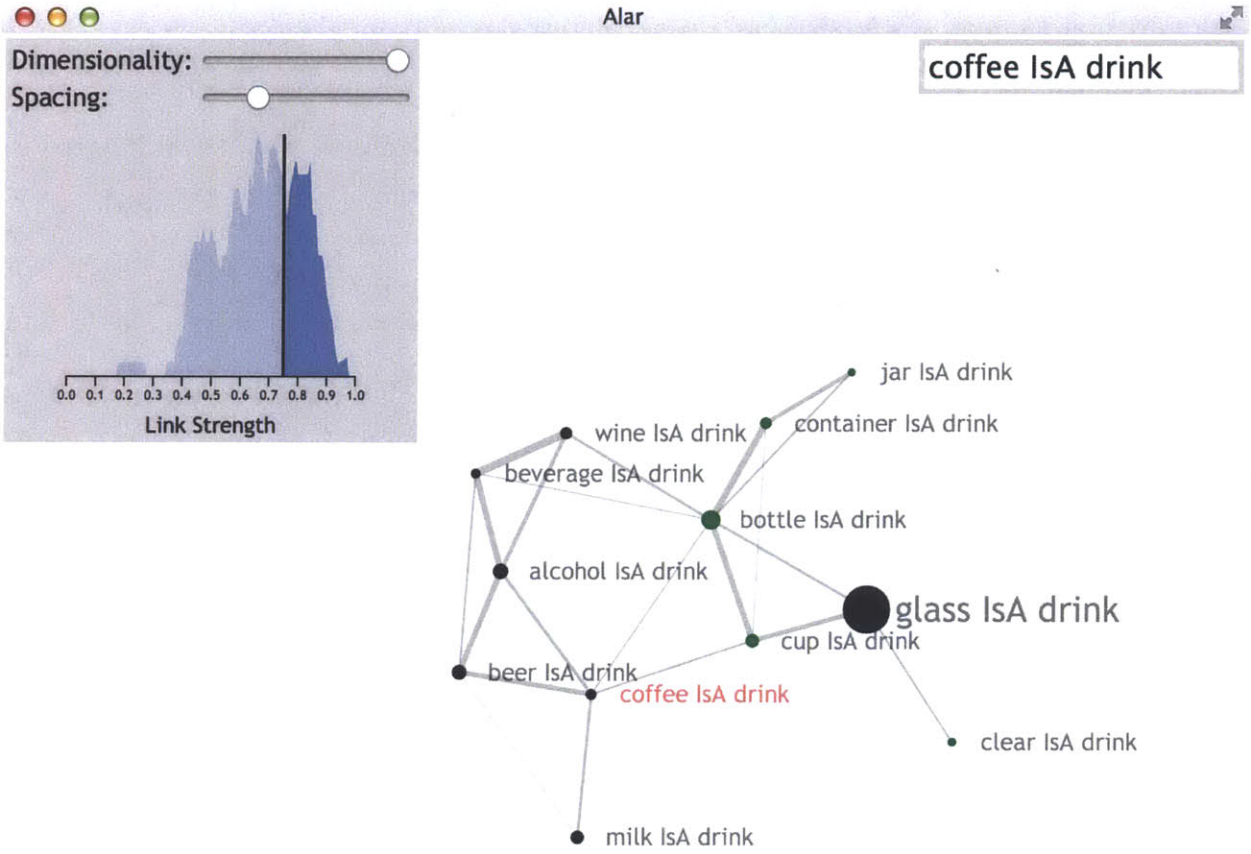

Figure 1-5: The interface shown after searching for "coffee IsA drink" and adjusting to improve readability.

Assertions have more information attached to them, however, so there is more which can be learned from the interface.

First, Sam can look for which assertions were inferred to be fairly true by looking at the size of the nodes. Larger nodes have been inferred to be more true and smaller nodes have been inferred to be less true.

Additionally, the color of the nodes illustrates whether or not the assertion was from the list of original facts or was inferred as a new combination of original facts.

In this example, Sam could see that "bottle IsA drink" was inferred to be fairly true, which in fact is not quite correct and Sam would want to understand why it was inferred so it could be corrected. Surveying the related assertions would show that "glass IsA drink" is very similar and an original assertion, but it is a false statement. Therefore, Sam can see it is very likely that this original assertion is causing incorrect assertions, such as "bottle IsA drink" to be inferred, and should be removed from the original list.

1.1.3 Adjusting The Dimensionality

Lastly, Sam may want to see how adjusting the parameters of the inference process affects the results. This can be done by adjusting the "Dimensionality" slider, in the top left of the interface.

This can be thought of as adjusting how liberal or conservative the inference process is. Conservative inference sticks closely to facts it already knows and only condones inference when there is a lot of evidence. Liberal inference, on the other hand, is quick to jump to conclusions even if there is a small amount of evidence. Sam would see this as the dimensionality slider is lowered and the graph becomes more and more clustered, as see in Figure 1-6. Sam can then adjust this parameter and see the visualization updated in realtime, allowing the desired setting to be found quickly.

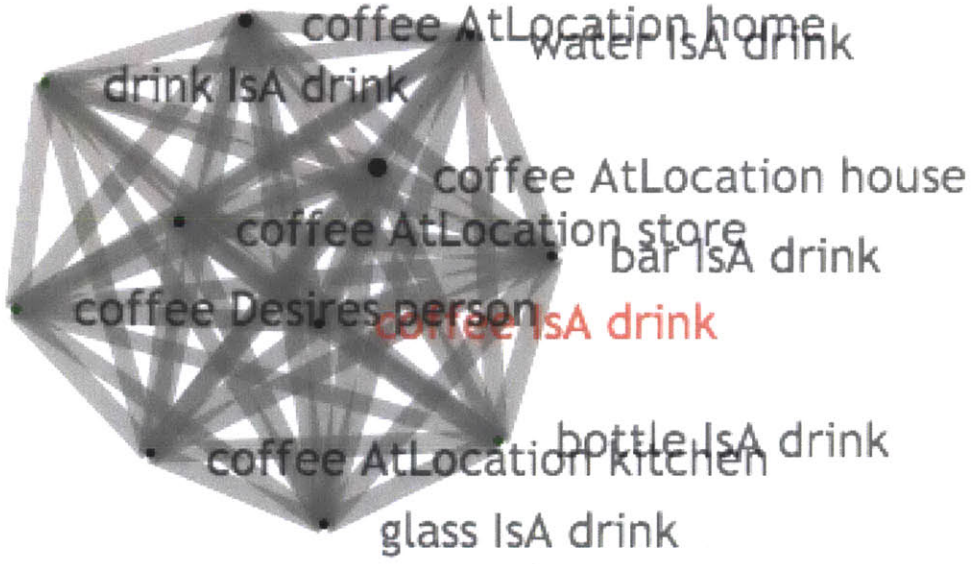

Figure 1-6: An illustration of the graph at lower dimensionality. At this dimensionality, the inference process is more liberal, meaning it infers everything to be much more related, resulting in a tightly clustered graph.

Now that the interface has been explained, the following chapters explain how it is made possible. First Chapter 2 explains the mathematical framework used by this thesis. Next, Chapter 3 explains how Alar was implemented, of which a major component was Celestrium, a library developed in parallel with Alar which is separately presented in Chapter 4. Following that, Chapter 5 discusses the results of usability testing of Alar's interface as well as a computational assessment of this thesis's novel metric for assertion relatedness. Chapter 6 discusses related work and Chapter 7 concludes.

Chapter 2

Mathematical Framework

The two central contributions of this thesis are being able to find the relatedness of assertions as well as being able to adjust the parameters of the inference in realtime. Achieving both of these required extending the AnalogySpace framework for inference [13].

Previously, there was no ability to compare how related assertions were because they are seemingly represented by a single value. This chapter explains how treating the summation which produced that value as a vector allows assertions to be compared for similarity, just as concepts are.

Additionally, previous implementations required using a fixed dimensionality. However, finding which value to use is extremely important in making a knowledgebase usable. This chapter explains that once the inference process has been run for a particular dimensionality, its results actually contain all the necessary information to proceed as if any lesser value of dimensionality was used.

This chapter begins by defining the different semantic entities which are involved in the process in Section 2.1. Next, it explains the original AnalogySpace framework for inference in Section 2.2. Finally, it concludes with this thesis's novel extensions to this framework in Section 2.3.

2.1 Semantic Entity Definitions

The inference process begins by taking in a list of simple facts. These facts must be of a particular form, and are composed of different semantic entities. This section introduces these semantic entities and explains the form of the facts which may be used for this inference process.

2.1.1 Concepts

Concepts are one of two primitive semantic entities used in the inference process. A concept, as the name implies, can be things like "water", "politics" or "happiness". They can effectively be thought of as nouns.

Concepts can be anything, but are typically normalized to a canonical form to avoid slight differences such as "drink" and "drinks", which would otherwise be treated as completely separate concepts. This type of normalization is certainly important, but not a focus of this work. Concepts are represented as strings.

2.1.2 Relations

Relations are the second of the two primitive semantic entities used in the inference process. As the name implies, they indicate some kind of relationship or connection between concepts.

Unlike concepts, relations are limited to a small, finite set of possibilities. See Table A.1 for the relations used by this work. Note, there is nothing intrinsically exceptional about the particular relations which were chosen. However, the fact that they are limited to a small set is the key property. This accomplishes a similar goal to that of normalizing the natural language of a concept - it ensures that facts are expressed in a similar enough language that there is at least some overlap between them. So in another context, a different small set of relations would serve just as well, so long as they capture the desired semantic relationships between the concepts.

2.1.3 Assertions

Now that concepts and relations have been established, the form of the facts which are used in the inference process can be explained. They are referred to as *assertions*, and are formed by an ordered triplet of (c_1, r, c_2), for concepts c_1, c_2 and relation r. For example, (coffee, IsA, drink) represents the fact which may be written in natural language as 'Coffee is a drink." This is different from (drink, IsA, coffee); the order matters.

Again, all the facts which initially form a KB must be of this form. In practice, they often put into a spreadsheet where each row is an assertion and each column is the first concept, relation, second concept and truth of that assertion, for a total of four columns.

2.1.4 Features

The last semantic entity is a *feature*. A feature is formed by a triplet of (d, r, c) for direction $d \in \{\text{left}, \text{right}\}$, relation r and concept c. A feature represents either the left or right half of an assertion, so there are two features which could be drawn from any one assertion.

Taking (coffee, IsA, drink) as the assertion, the two features it could produce are (right, IsA, drink) and (left, IsA, coffee). We refer to them as the right and left features, respectively.

Furthermore, a concept can be combined with a feature to make an assertion. (right, IsA, drink) combined with "coffee", forms the assertion (coffee, IsA, drink). It essentially fills in the missing concept of the feature.

With these entities now defined, the next section explains the original framework for inference on top of which this thesis builds.

2.2 AnalogySpace Framework for Inference

This thesis builds heavily on top of AnalogySpace. In the previous section, the necessary terminology was introduced to be able to adequately describe this process. This section now explains the original inference process presented in AnalogySpace. This is in preparation for Section 2.3, which introduces the novel extensions of this thesis.

2.2.1 The Assertion Matrix

The entire goal of the inference process is to be given some facts and then to infer others, as well as determine the relatedness of their semantic entities. This original list of facts and the results of the inference process are often referred to as a Knowledgbase (KB.) To do this, an alternative representation of the KB's original assertions are used. This is the assertion matrix.

This representation is equivalent to the list of assertions in that one can be constructed from the other, but the assertion matrix has the benefit of being able to be manipulated by mathematical operations. Both these ideas summarize this entire framework at the highest level. Assertions are transformed into a matrix. Mathematical operations are used to create a similar matrix, which is turned back into assertions to reveal the results of the inference in a semantic sense, not just mathematically. This process is all explained in detail in the following sections, but first, the assertion matrix must be explained.

In short, each row of the matrix corresponds to a unique concept and each column corresponds to a unique feature. We often refer to this, given a concept and feature c and f, as c's row and f's column. The matrix starts as a zero matrix. Then, for each assertion, $a = (c_1, r, c_2)$ with right and left feature f_r, f_l, two cells are given a value of 1 to indicate they are true or -1 if the assertion is labeled as false. They are the intersections of each of c_1's row and f_r's column as well as c_2's row and f_l's column. Both these cells represent a, which is why both are entered in this way.

Note, this is a very dubious proposition, as nothing is mathematically constraining these cells to remain equal in the inference process, so it results in two different inferred truth values for an assertion. This work uses both entries to find related assertions, but only uses the entry for the first concept and right feature to find an assertion's truth.

All original assertions can be true or false. Assertions marked as true will enter a 1.0 into their cells in the assertion matrix. Assertions marked as false will enter a -1.0 into their cells in the assertion matrix. However, this is often an extremely sparse matrix. The next section explains how the empty cells in the matrix are filled in to produce inferred truth values for

all assertions that were originally specified.

2.2.2 Performing Inference Through SVD

As stated, forming an assertion matrix from the list of assertions provides exactly the same information, or in other words, no new information. So how is inference performed?

Before describing the technique itself, I'd first like to frame the form of the result of the technique. In essence, it aims to produce a matrix which is an approximation of the original assertion matrix, filling in the zero values with nonzero values. This is equivalent to, semantically speaking, inferring the truth of assertions which were not explicitly specified in the original list of assertions but are composed of the same concepts and features. Additionally, this result provides the ability to determine the similarity of any two concepts or features involved.

With that in mind, the actual technique which is used to approximate the original assertion matrix is to take its truncated singular value decomposition (SVD). First, I explain the complete SVD, then I describe a truncated SVD.

The SVD of an original assertion matrix, A, results in matrices U, Σ, V st.

- $A = U\Sigma V^T$
- U's columns are the orthonormal eigenvectors of AA^T.
- V's columns are the orthonormal eigenvectors of A^TA.
- Σ is a diagonal matrix whose entries squared are the eigenvalues of AA^T and A^TA in decreasing value.

Due to the size of A in practice, it is often infeasible to compute the full SVD, which is why a truncated SVD is used instead.

A truncated SVD of A only considers the first, or equivalently highest, r singular values of the SVD, zeroing out the rest. This effectively discards all but the first r columns of U and V as well.

This results in

$$A \approx \tilde{A} = U\Sigma_r V^T$$

$\tilde{A}$ is exactly the approximation of A the inference process aimed to find! Every other part of this work revolves around this one operation.

Next, we conclude the original AnalogySpace inference process with how to interpret the results of this core operation.

2.2.3 The Results of the Inference Process

Finally, given $A \approx \tilde{A} = U\Sigma_r V^T$, what does this actually provide?

First, it provides inferred truth values of not only the original assertions, but also of any assertions composed of any combination of concepts and features found in the original assertions. These are found by locating the cell in $\tilde{A}$ corresponding to a given assertion and using it's value as the inferred truth of that assertion.

Second, it provides the ability to compare the relatedness of any two concepts. As stated above, each row corresponds to a concept, so multiplying AA^T yields the concept similarity matrix, because each row and column in this product corresponds to a concept, so for any cell, this is the inferred similarity of it's row and column's concepts.

At last, we have some semantically meaningful results from this process. Starting with a list of facts, we can infer the truth of related facts as well as suggest the semantic relatedness of their concepts.

With this process in mind, the next section presents this thesis's novel extensions to this framework for inference.

2.3 Novel Extensions to AnalogySpace

With an understanding of the basic AnalogySpace framework for inference, this section explains this thesis's novel contributions to the framework.

The contributions begin with the notion of *semantic entity vectors*. From this abstraction, not only are concepts able to be compared for semantic relatedness, but so to are features and even assertions. Additionally, this abstraction also provides the ability to find the results of the entire process as if any number of singular values less than r have been used, without having to recompute any SVD.

This section first presents the concrete definition for concept, feature and assertion vectors. Then, it is explained how the relatedness and truths of assertions are found from these representations. Finally, it is explained how the results change as the desired number of singular values is varied.

2.3.1 Concept Vectors

The first type of semantic entity vector is the concept vector.

As a core semantic entity of this inference process, we'd like to be able to gain some insight about concepts as a result of the inference process. Finding the similarity of concepts was the central focus of AnalogySpace and this work's novel abstraction achieves the same results as in AnalogySpace, but with the added benefits described above.

To start, recall that each row in $\tilde{A}$ represents a concept. We then define the relatedness of two concepts as the inner product of their rows. Then, the concept similarity matrix, $\tilde{A}\tilde{A}^T$ defines the similarity of every pair of assertions. Specifically, an entry i, j in $\tilde{A}\tilde{A}^T$ is the similarity of the concepts represented in rows i and j in $\tilde{A}$. However, there is a much more efficient way to find concept similarities than computing $\tilde{A}\tilde{A}^T$.

$$\begin{aligned}
\tilde{A}\tilde{A}^T &= (U\Sigma V^T)(U\Sigma V^T)^T \\
&= U\Sigma V^T V\Sigma^T U^T \\
&= U\Sigma\Sigma^T U^T \\
&= (U\Sigma)(U\Sigma)^T
\end{aligned}$$

This allows computation of concept similarities much more efficiently in practice for several reasons. First, the matrix $\tilde{A}$ need not be constructed, let alone multiplied. In practice, the U, Σ, V are found but never fully reconstructed into $\tilde{A}$. Additionally, AnalogySpace's implementation creates a matrix from the lazily evaluated product of $U\Sigma$ and its transpose, so again, the full multiplication does not need to be performed.

This work introduces another level of abstraction - the notion of a concept vector.

A concept's vector is defined to be its row in $U\Sigma$, which is

$$(\sigma_1 U_{i,1},\ \sigma_2 U_{i,2},\ \ldots\ , \sigma_k U_{i,r})$$

Where i is the concept's row in U and σ_d is the d^{th} singular value.

This is a simple and powerful abstraction for the following reasons.

1. It is a concise representation of a vector. Note that $\tilde{A}$ represents each concept as a row vector in as many dimensions as there are features, m. $U\Sigma$ represents each vector in r dimensions, which in practice can be three orders of magnitude smaller than m.

2. It preserves the pairwise similarities found using $\tilde{A}\tilde{A}^T$. This is evident because, as shown above, $\tilde{A}$ and $U\Sigma$ are defined to have the same product with their transpose. This means the inner product of rows i, j in $\tilde{A}$ is the same as in $U\Sigma$, which is exactly the operation used to determine which concepts are similar to each other. In short, it produces the exact same results as in AnalogySpace.

In summary, by explicitly representing concepts as vectors, we gain the above advantages and as explained next, the same advantages apply to features as well.

2.3.2 Feature Vectors

It is a relatively straightforward progression from concept vectors to feature vectors. Simply consider swapping the roles of concepts and features, or in other words, using the transpose of $\tilde{A}$ for all operations.

If $\tilde{A} = U\Sigma V^T$ then

$$\begin{aligned}\tilde{A}^T &= (U\Sigma V^T)^T \\ &= V\Sigma^T U^T \\ &= V\Sigma U^T\end{aligned}$$

U and V have just switched roles, so here is the definition.

A feature's vector is defined to be its row in $V\Sigma$, which is

$$(\sigma_1 V_{j,1},\ \sigma_2 V_{j,2},\ \ldots\ , \sigma_r V_{j,r})$$

Where j is the feature's row in V and σ_d is the d^{th} singular value.

Another way this can be seen is through a similar decomposition as before.

$$\begin{aligned}\tilde{A}^T\tilde{A} &= (U\Sigma V^T)^T(U\Sigma V^T) \\ &= V\Sigma^T U^T U\Sigma V^T \\ &= V\Sigma^T\Sigma V^T \\ &= V\Sigma\Sigma^T V^T \\ &= (V\Sigma)(V\Sigma)^T\end{aligned}$$

So just as $U\Sigma$ perfectly preserves the pairwise comparisons of concepts, $V\Sigma$ perfectly preserves the pairwise comparisons of features. Additionally, reducing the number of singular values again just removes the last columns of $V\Sigma$, one per reduced singular value. So in summary, feature vectors are analogously defined to concept vectors. Assertion vectors, however, are not as straightforward and are explained next.

2.3.3 Assertion Vectors

It is relatively straightforward to see the vector representations of concepts and features, as they are just rows in matrices already presented or easily found. Assertions, however, have no immediate vector representation, just a single value: their inferred truth. However, if this single value is decomposed through the linearly separable model view of and SVD, it can be seen to be the following sum for an assertion with first concept and right feature at row and column i and j in U and V, respectively.

$$\sigma_1 U_{i,1} V_{j,1} + \sigma_2 U_{i,2} V_{j,2} + ... + \sigma_k U_{i,r} V_{j,r}$$

Using each element in the sum as a dimension in a vector, it leads to the following definition.

Given an assertion with first concept c_1 and right feature f_r, its vector is

$$(\sigma_1 U_{i,1} V_{j,1},\ \sigma_2 U_{i,2} V_{j,2},\ ...\ , \sigma_r U_{i,r} V_{j,r})$$

Where i is c_1's row in U, j is f_r's row in V and σ_d is the d^{th} singular value.

This is motivated semantically in several ways.

1. This vector represents the change in truth value of the assertion at each dimension. One would think that similar assertions would be inferred to be more true or false in a similar fashion due to the contribution of each new singular value.

2. Just as a feature and a concept can be combined to form an assertion, their vectors' inner product can be decomposed into this vector, less an extra factor of Σ.

While these were just the motivations, note that Section 5.2 shows this abstraction to be effective in practice.

2.3.4 Inferring Similarity Using Semantic Entity Vectors

AnalogySpace was able to find the relatedness of just concepts. By representing concepts, features and assertions as vectors, we can apply the following definition to all of them, enabling the relatedness of any pair of similarly typed semantic entities to be found.

The similarity of any two similarly typed semantic entities with vectors v_1, v_2 is defined to be their cosine similarity.

$$\text{entity similarity} = \frac{v_1 v_2}{|v_1||v_2|}$$

This is a beneficial definition for the following reasons.

1. It treats all entities equally. With no normalization, some if a particular concept or feature has far more assertions about it than another one, the sheer size of the vector results in it being perceived as similar to many other concepts when an unnormalized inner product is used. When normalized in this way, each vector is made to be of unit length, thereby treating each one with equal weight.

2. It bounds the similarity of two concepts to be in $[-1, 1]$, where -1 is exactly dissimilar and 1 is exactly similar. Knowing this bound is very helpful in understanding the absolute similarity of any two concepts.

In short, semantic entity vectors allow this straightforward definition of similarity for concepts, features and assertions. Next, how to find the inferred truth of an assertion given its vector is explained.

2.3.5 Inferring Truth Using Assertion Vectors

AnalogySpace provided a straightforward method to find the inferred relatedness of an assertion; simply uses its entry in $\tilde{A}$. However, this scalar value has been decomposed to form a

vector to represent the assertion. Fortunately, it is straightforward to compute an assertion's truth when given its vector.

The truth of an assertion with vector, v, considering r singular values is

$$\text{assertion truth} = \sum_{d=1}^{r} v[d]$$

This is exactly the inferred truth which would have been determined in AnalogySpace, but is now explained within the new context of being given an assertion's vector.

With the process defined to find these results given these semantic entity vectors, the next section explains how the vector for any semantic entity could be efficiently found as if the SVD had been recomputed for any number of singular values.

2.3.6 Efficiently Computing Results for Fewer Singular Values

A benefit of semantic entity vectors is that given a vector produced from the inference process when considering r singular values, the vector for the same semantic entity can be found as if the process had been repeated for any number of singular values $r' \leq r$, without having to recompute any SVD. This section explains how and why this is possible.

The definition is extremely straightforward, and we'll see why this is the case.

A semantic entity's vector for $r' \leq r$ singular values is simple the first r' components of it's vector found using r singular values.

$$v_{r'} = v_r[:r']$$

Where v_r is the semantic entity's vector found considering r singular values and $v_{r'}$ is the semantic entity vector when considering r' singular values.

This definition is extremely straightforward, but why does it work? First, semantic entity vectors are constructed to behave in certain ways. To remain consistent with the original

AnalogySpace framework, let's examine how this definition produces the same results for assertion truth and concept similarity.

In AnalogySpace the truth of an assertion is simply its cell's value in $\tilde{A}$. Say $\tilde{A}_r$ was constructed using r singular values. We've already shown that for any entry i, j in the $\tilde{A}_r$, its value is the summation $\sigma_1 U_{i,1} V_{j,1} + \sigma_2 U_{i,2} V_{j,2} + ... + \sigma_r U_{i,r} V_{j,r}$. We've also already shown that treating each element in this sum is exactly how an assertion's vector is defined, so varying the number of singular values varies both this summation and the assertion vector consistently, always maintaing the invariant that the sum of an assertion's vector is its inferred truth value in $\tilde{A}$.

Now for concept similarities. In AnalogySpace, the similarity of two concepts represented by rows i, j in $\tilde{A}$ is the value of entry i, j in $\tilde{A}\tilde{A}^T = (U\Sigma)(U\Sigma)^T$. This entry is then exactly the inner product of the rows i and j in $U\Sigma$, which is defined to be the summation $\sum_{d=0}^{r} U_{i,d} U_{j,d} \sigma_d^2$. Using concept vectors, we've defined the similarity of two concepts to be the inner product of their vectors, which is exactly the same summation. Again, as the number of singular values changes, this summation changes as does the semantic entities' vectors' lengths to maintain the invariant that the inner product of two concept's vectors is their inferred similarity as defined in AnalogySpace.

So, we can see that semantic entity vectors produce results consistent with the AnalogySpace framework. Additionally, we've seen that the semantic entity vectors can be efficiently computed as if any number of singular values less than the original amount were used and then used to find their inferred truth and similarity.

Finally, this chapter concludes by characterizing the behavior of these quantities as the number of singular values changes.

2.3.7 Expected Behavior When Varying the Number of Singular Values

We can efficiently compute the truth of assertions and similarity of all semantic entities for any number of singular values. But how do these results actually change when the number

of singular values is adjusted? It certainly varies with the actual data, however this chapter suggests the expected behavior, which is that as the number of singular values increases, semantic entities become less related. Additionally, as the number of singular approaches $rank(A)$, original assertions are inferred to be more true and other assertions are inferred to be less true. Here's why.

First, note that considering only the first singular value, each vector only has 1 dimension. Then, normalized to be of unit length, this single dimension's value must be either 1 or -1. Thus, any two entities are interpreted to be either exactly similar or exactly dissimilar.

As more singular values are considered and the entities' vectors have more dimensions, this effect would presumably be lessened because to remain perfectly similar or dissimilar, the additional entries into their vectors would have to match perfectly, which is unlikely. As more dimensions are added, more variation is likely introduced and therefore, the concepts are interpreted to be less and less similar. This results in the following hypothesis.

The absolute value of the similarity of two entities varies inversely with the number of singular values.

This is not only supported by the above intuition, but has been repeatedly observed in practice. Now, we see that the opposite is true of assertion truth values, but this makes sense semantically.

An assertion's vector is the change in its inferred truth value with each additionally considered singular value. With this in mind several hypotheses can be made about the behavior of different types of assertions inferred truth values.

First, for an assertion which was originally in the matrix, if r was the rank of the original assertion matrix, A, the inferred truth value would be exactly 1, by definition, since $\tilde{A} = A$ in this instance. Similarly, for assertions which were not originally in the matrix, their values must be 0, in this case. In between however, it is assumed the values are generally positive, as the matrix starts out with mostly positive numbers in it. This leads to the following hypotheses.

The inferred truth value of an assertion originally in the KB approaches 1 as the the number of singular values approaches the rank of the original assertion matrix.

The inferred truth value of an assertion not originally in the KB approaches 0 as the number of singular values approaches the rank of the original assertion matrix.

Note that assertion truth values of -1 and 1 indicate the assertion is false and true respectively. Therefore we interpret 0 as being unknown.

Chapter 3

Implementation

With an understanding of what Alar presents to the user as its interface as well as the mathematics behind it, this chapter now details how it was implemented. It separates the implementation into two parts, beginning with the frontend and ending with the backend. Alar's implementation can be found at `http://www.github.com/jdhenke/alar`.

3.1 Alar's Frontend

This section describes the important techniques of how Alar's interface can provide the information and responsiveness it is responsible for.

3.1.1 Use of Celestrium

Firstly, Alar uses Celestrium, a CoffeeScript library developed in parallel with Alar for use by Alar but it is also usable in many other contexts. Chapter 4 details the implementation of Celestrium, however for the purposes of this chapter, here is what Celestrium does for Alar's frontend implementation.

Firstly, it provides a framework to maintain an underlying model of the graph's nodes and links. Additionally, it automatically propagates any changes made to the model to any listeners. Lastly, it is responsible for rendering the graph itself and automatically re-rendering

when any changes are made to the model.

This ultimately allows the implementation specific to Alar to only concern itself with how the underlying values of the model should change, and Celestrium will automatically render the graph to reflect the new values.

3.1.2 Enabling Realtime Variance of SVD Rank

One of the novel features of Alar is that a user can adjust the rank of the SVD of the inference process and see the visualization adjust to correctly display the results at the new rank, all in realtime. Alar accomplishes this by essentially combining two techniques. The first is leveraging the mathematics of the truncated SVD to efficiently find results for all possible values of k. The second is that a polynomial can then be constructed to interpolate the results as k varies. How these are constructed is left for Section 3.2, but given they can be constructed, here is how the frontend leverages them.

First, I clarify what results can mean, or in other words, what aspects of the interface can change. Both the truth of an assertion and the relatedness of any two semantic entities are aspects of the interface which may change with the rank.

Therefore, whenever a node or link is added to the graph, the server doesn't provide the static value for the current rank, it provides the coefficients of the polynomial in the rank for that value. Upon initially receiving this information, the client then reconstructs the polynomial in Javascript so it can be called with the value of the current rank and used to update the value of the model, which Celestrium then propagates and renders in the interface immediately.

This technique was chosen for a variety of reasons. While an integer number of singular values can be used, it is desirable for the interface to smoothly update itself. So rather than jumping from one integer number of singular values to the next as the user adjusts the slider, the polynomial can be evaluated at any real value, providing a visually smooth adjustment of the interface. Additionally, the polynomial is constructed in such a way as to provide exact answers at each integer number of singular values the user selects. This keeps the visualized

results true to the results of the underlying inference process. Lastly, reconstructing the function on the client means the server does not need to be contacted at all when the user adjusts the rank slider. This allows the interface to instantaneously update itself as the user adjusts the slider.

3.2 Alar's Backend

This section describes the main approaches Alar's server takes to efficiently provide the frontend with the information it needs.

3.2.1 Preparing a Knowledgebase

Alar can visualize inference over any knowledgebase but first, the knowledgebase must be prepared. This essentially involves precomputing the results of the SVD.

Alar provides a python script to prepare the knowledgebase given a CSV file of the assertions. The CSV file should have one assertion per line, with the first concept in the first column, the relation in the second column, the second concept in the third column and the truth of the assertion in the fourth column. The truth should be 1 if true and -1 if false.

From this list of assertions, the preparation script creates the assertion matrix and computes its truncated singular value decomposition, then saves the results to disk. This is simply to make the web server startup faster because this computation would have to be done on every server startup.

Ultimately, the preparation script allows Alar to quickly start and serve its interface for exploring any KB.

3.2.2 Finding Similar Semantic Entities

A central function of Alar's interface is to find related semantic entities to a given, root semantic entity. In Chapter 2, it is explained how to find the relatedness of any two semantic entities; simply take the cosine similarity of their vectors. To find the relatedness using any

number of singular values, simply take the inner product of their truncated vectors. Now, however, Alar needs to find the relatedness of every other semantic entity to the given entity and return the ones which are the most similar.

Alar maintains `u, s, v` which are the results of the truncated singular value decomposition of the original assertion matrix. Note that `s` is a one dimensional array which is the diagonal of Σ in the SVD. These are `numpy` arrays, which allows many convenient options, such as $O(1)$ slicing. For example, `u[:,:r]` returns an object which may be treated as a new array containing all the rows of `u` and only the first `r` columns. However, this new object doesn't copy all that data, it is merely referencing the underlying data in `u`, so it is a very efficient operation. Note that modifying `u[:,:r]` will modify `u`, but all operations in Alar are read only, so this is not an issue.

Let's begin with concepts. Note that related features are found in the same way, only using `v` instead of `u`. When computing related concepts to some concept `c` with row `i` in `u` considering `r` singular values, note that each row in

```
concept_vecs = (u*s)[:,:r]
```

is exactly each concept's vector. Then, `c`'s vector is just `concept_vecs[i]`. Finally, each concept's similarity is just its entry in

```
concept_sims = concept_vecs.dot(c)
```

So we can just use the concepts with the maximum values in `concept_sims`.

Assertions are not quite as straightforward. First, note that an assertion with concept and feature indices `i` and `j` is found with `vec = u[i] * s * v[j]`. For `numpy` arrays, the `*` operator induces broadcasting, which in this case results in an array `vec` such that `vec[x] = u[i,x] * s[x] * v[j,x]` for all `x`, which is exactly the definition of an assertion vector.

However, which assertions should be considered? In practice, it is infeasible to consider all assertions, because there are too many to consider and return a result to the user in a reasonable amount of time. As a compromise, only assertions which share a concept or

feature are considered. We create an array with each row as an assertion vector, just as with concepts, in the following way:

```
a_vecs_same_feature = u*s*v[j]
a_vecs_same_concept = v*s*c[i]
all_a_vecs = np.concatenate(a_vecs_same_feature, a_vecs_same_concept)
a_vec = u[i]*s*v[j]
sims = all_a_vecs.dot(a_vecs)
```

From this, the assertions corresponding to the top items in `sims` are the most related assertions to return.

This concludes the explanation of how similar semantic entities are found for concepts, features and assertions. However, once found, the frontend requires not only their relatedness for a particular rank of the SVD, but a function of the current rank. This is explained next.

3.2.3 Constructing Polynomials in r

As explained in Section 3.1.2, it is the responsibility of the backend to provide not only scalar values for the relatedness of two semantic entities or the truth of an assertion, but to provide a polynomial in the rank of the SVD to the frontend.

Therefore, when the backend receives such a query, it creates the polynomial in the following way with a very succinct, functional style. We will use the example of finding the coefficients of the polynomial representing the similarity of two concepts.

```
def get_concept_similarity_coeffs(self, c1, c2):
  v1, v2 = map(self._get_concept_vector, (c1, c2))
  return self._interpolate(lambda d: norm_dot(v1[:d], v2[:d]))


def _interpolate(self, f):
    values = map(f, xrange(1, self.rank + 1))
    return np.polyfit(range(self.rank), values, self.rank - 1)
```

In words, this finds the concept vectors of the given concepts. Next, it creates values for their normalized dot product considering d dimensions for $d \in [1, k]$. Finally, it returns the coefficients of the polynomial fit to the values just created.

The order of the polynomial is the one less than the number of values, which is the rank of the SVD. This was chosen because there is exactly one such polynomial and its value at each integer value is exactly the similarity found by the inference process.

3.3 Design Decisions

This section highlights the most notable design decisions of Alar, providing the context and justification for each.

3.3.1 Use of Divisi2

Divisi2 is AnalogySpace's python implementation and provides the functionality described as part of the original AnalogySpace framework for inference. The decision was made to not use any of Divisi2's code and to build Alar's backend from scratch for the following reasons. First and foremost, Divisi2 was built several years ago and at the time a different numerics library was more readily usable. Unfortunately, patently incorrect results were experienced in using this library under certain conditions and it was so entrenched in Divisi's code base, no clear port to scipy, the new standard package for sparse matrix manipulation in python, was seen.

Additionally, while both Divisi2 and Alar use the same basic AnalogySpace framework for inference, Alar's approach is substantially different through its use of semantic entity vectors and so the primary abstractions greatly differed in the codebases.

3.3.2 Use of D3

Many Javascript visualization libraries exist, so the choice of D3 [4] merits some justification. D3 was chosen for the following reasons. First, it is flexible. It does not tie the interface to

any particular visualization in fact, it is merely used to facilitate binding data to elements in the DOM. Secondly, there are many working examples available which use D3. Lastly, D3 comes with the graph visualization library used by Alar's interface right out of the box. This allowed the organic, dynamically adjustable graph visualization and clustering that is the central component of Alar's interface, so this feature in particular was a driving force in choosing D3 as the primary Javascript visualization tool used by Alar.

The next chapter explains Celestrium, which was a core part of Alar's interface implementation and is also an independent contribution of this thesis, usable in other contexts as well.

Chapter 4

Celestrium

Celestrium is a CoffeeScript library which enables developers to easily create interfaces for exploring large, graph-based data sets. Celestrium was developed as part of this thesis for two reasons. First, it is used by Alar's frontend to create the graph of related semantic entities. Second, by keeping Celestrium data set agnostic, it is now a useful contribution by itself, and it is mostly within this context that this chapter presents Celestrium.

This chapter argues the following points, each in their own section. First, from a design perspective, Celestrium employs a unique plugin architecture. It is an MVC framework [10] built specifically for dynamically exploring large graph based data sets. Additionally, its plugin dependencies are made explicit so that plugins are created in the correct order. Traditional plugin architectures assume there are no dependencies between the plugins, but in this case, the dependencies between the plugins are critical. This allow plugins to be dynamically created in the correct order at runtime as opposed to forcing the developer to manage the logistics of the CoffeeScript compilation process. This architecture is explored and justified in Section 4.1. Second, Celestrium is itself a concrete implementation and provides a lot of functionality itself. This is illustrated with an explanation of the plugins which come with Celestrium in Section 4.2. Last, to illustrate its ease of use, an example implementation is explained in Section 4.3.

4.1 Plugin Architecture

Celestrium's plugin architecture stemmed from its target environment, which is a user interface in which many optional components exist and interact with each other. These components will be called plugins. Therefore, being able to access the instance of any other plugin is very important.

Furthermore, if these plugins depend on each other to operate, it is very important they are instantiated in an order which defines each plugin's dependencies before defining that plugin so that its dependencies are immediately available upon instantiation.

Celestrium addresses these issues by requiring each plugin to explicitly enumerate its dependencies. It can then analyze the graph of plugin dependencies and instantiate them in a proper order. Each of these topics are detailed in the rest of this section.

4.1.1 Specifying Dependencies

As complicated as dependencies may seem, Celestrium makes it easy to specify and access a plugin's dependencies.

Celestrium plugins are defined as CoffeeScript classes. Each plugin must have a unique *uri* string. Each plugin specifies its dependency *needs* by a dictionary, mapping an attribute name to the uri of the plugin whose instance will be made available to that plugin under that attribute name. Both of these are class attributes.

For example, the Data Provider plugin definition dictates its *uri* and *needs* as follows.

```
class DataProvider


  @uri: "DataProvider"
  @needs:
    graphModel: "GraphModel"
    keyListener: "KeyListener"
    nodeSelection: "NodeSelection"
```

```
constructor: () ->
  @keyListener.on "down:16:187", () =>
    @getLinkedNodes @nodeSelection.getSelectedNodes(),
    (nodes) =>
      _.each nodes, (node) =>
        @graphModel.putNode node if @nodeFilter node

# remaining implementation omitted.
```

This states that other plugins may depend on DataProvider by using it's *uri*, `DataProvider`. Then, in any function DataProvider defines, it may access the instance of the plugin with uri `GraphModel` by using the attribute `graphModel`. In CoffeeScript, this is simply `@graphModel`, as seen in the last line of the constructor shown above. Note that the dependencies are made available even in the constructor of the object. How this is made possible is explained next.

4.1.2 Ordering Instantiation of Plugins

As seen in the above example, the DataProvider plugin depends on having access to the instance of the GraphModel in its constructor. Therefore, the GraphModel plugin should instantiated before the DataProvider plugin.

More generally, all of a plugin's dependencies must be instantiated before that plugin is itself instantiated. Celestrium achieves this by first forming a DAG of dependencies, where each node represents a plugin and there exists an edge $a \rightarrow b$ if plugin a depends on plugin b. Celestrium then instantiates plugins in reverse topological order. Cyclical dependencies are forbidden in Celestrium because a plugin may need access to the instance of a plugin it depends on even in its constructor.

Figure 4-1 illustrates the DAG of dependencies formed by some of the default plugins which come with Celestrium. For example, because GraphView depends on GraphModel, GraphModel must be instantiated first. This figure was actually generated using Celestrium,

which is the sample implementation explained in 4.3.

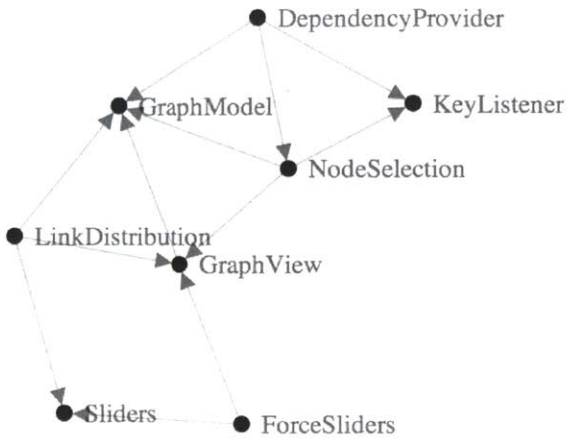

Figure 4-1: An example dependency graph of Celestrium plugins.

4.1.3 Justification of Forbidding Circular Dependencies

Many alternative designs were considered in the process of building Celestrium. Of central importance was whether or not to allow circular dependencies. Ultimately, no circular dependencies are allowed as a DAG of dependencies is required and thus cannot have cycles. The decision merits an explanation of both sides of the argument. If Celestrium allowed circular dependencies, it would be a less restrictive interface for the developer and allow more configurations of plugins. However, even if circular dependencies were allowed, an instance of each plugin must be created in some linear order and if that is so, there is no way for Celestrium to guarantee a plugin's dependencies are defined for it. Therefore, Celestrium chose the more restrictive design with better guarantees for the developer.

4.2 Builtin Plugins

Celestrium comes with many plugins already defined, each of which provides functionality common to interfaces for exploring graphs. A complete list can be found in table A.2. This

section describes the most important plugins that Celestrium provides.

4.2.1 GraphModel

GraphModel stores the state of the graph currently being rendered to the user. Specifically, this is comprised of a set of nodes and a set of links such that the source and target of each link is contained within the set of nodes. The sets of nodes and links are maintained in a Backbone.js Model. This allows other plugins to register listeners which are called any time these sets change.

A subtle point of the implementation is that Celestrium in fact triggers these changes. Backbone's spec says that it will trigger events when any of a model's attributes change. Consider the following snippet being called for a Backbone Model with attribute "nodes".

```
data = @get "nodes"
data.push datum
@set "nodes", data
```

This does not in fact cause Backbone to trigger a change for the nodes attributes. This is because in testing for a change, it tests equality of the current value and the new value. Because `@get` returns the actual underlying object, `data.push` modifies `data`, which is in fact the internal object stored by the model. So when `@set` is called, it is comparing equality of the same object, so no change is detected and no events are fired. Therefore, Celestrium manually triggers these events

```
@trigger "change:nodes"
@trigger "change"
```

4.2.2 GraphView

GraphView is the plugin responsible for rendering the Graph based on the GraphModel. It listens for changes in the GraphModel and re-renders itself based on the new state of the GraphModel.

GraphView makes extensive use of D3.js. D3 was chosen for several reasons. First, it is a very flexible framework. In fact, it doesn't tie the user to any particular type of visualization, it merely presents a powerful paradigm of binding data to elements in the DOM. Second, it is widely used and there are many working examples available. Lastly, D3 comes with a Force Layout for graphs, which powers the realtime graph layout visualization present in Alar's interface.

Additionally, GraphView allows other plugins to augment the rendering of the graph by exposing its D3 selection of the current set of nodes and links. A selection is an abstraction introduced by D3 and essentially facilitates manipulation of the DOM elements portraying the graph. For example, Alar allows nodes to be "highlighted" when clicked. The plugin responsible for this feature accesses the current node selection through the instance of the graph view plugin to register event handlers on all nodes and to appropriately color them. In other words, GraphView doesn't provide any of this functionality, however it is designed in a way which, thanks to D3, allows other plugins to augment the graph very easily.

GraphView uses Scalable Vector Graphics (SVG) to visualize the graph. This was chosen over paintable alternatives such as Canvas in HTML5 because it fluidly adjusts when zooming and allows existing graphics to be modified very easily, such as when the user adjusts the dimensionality and link strengths must change. It is also much easier to draw text in SVG.

4.2.3 LinkDistribution

The last of the core plugins to be described is the LinkDistribution plugin. This provides the visual of the distribution of link strengths as well as the ability to filter which links are shown in the graph. This is implemented using D3 and SVG just as with the graph. The aspects of the implementation worth mentioning are how the distribution to be shown is derived as well as how the plugin interacts to update the graph in realtime.

To start, here is how the distribution is calculated. This distribution is actually a "smoothed" histogram of link strengths. Specifically, an actual histogram is created with 100 bins. This number was simply chosen by assessing the results of the display in practice.

A higher number would yield a smoother chart while a lower number would result in a very coarse grained chart. Then, the smoothed version of the histogram is created in the following way. The value at each bin is the average of the 10 closest original bins. By averaging over the nearby bins, this bin's value is representative of the number of links with strengths close to it and avoids the issue of having many empty bins that have many nearby values.

Next, how the plugins update the graph in realtime is a great example of the ease with which Celestrium's plugins interact. Essentially, whenever the threshold is moved by the user, the following line is all that is necessary to trigger how the update is rendered in the Graph.

```
@graphView.getLinkFilter().set("threshold", @x.invert(newX))
```

In words, LinkDistribution depends on GraphView, so it is accessible as the `@graphView` attribute. Next, GraphView exposes the filter it uses to determines which links are displayed. Finally, this plugin simply adjusts the threshold by translating the value of the slider in the interface to the threshold of relatedness two nodes must have to merit a link between them. GraphView has a listener established for when this threshold is changed and re-renders itself, using the new threshold.

This concludes the discussion of some of Celestrium's core plugins. Next, an example of a full implementation is explained.

4.3 Example Implementation Using Celestrium

In this section, the details of an example implementation are explained. The graph that will be visualized is actually related to Celestrium - it is the DAG of plugin dependencies which are used to create the visualization itself, as seen in Figure 4-1.

First, the main script is as follows.

```
$ () ->
  celestrium.init
    "DependencyProvider": {}
    "Layout":
      el: document.querySelector "body"
    "KeyListener":
      document.querySelector "body"
    "GraphModel":
      nodeHash: (node) -> node.text
      linkHash: (link) -> link.source.text + link.target.text
    "GraphView": {}
    "NodeSelection": {}
    "Sliders": {}
    "ForceSliders": {}
    "LinkDistribution": {}
    "NodeSearch":
      local: _.keys celestrium.defs

  , (instances) ->
    for uri, instance of instances
      instances["GraphModel"].putNode
        text: uri
```

This tells Celestrium which plugins to use in this visualization and provides any constructor parameters they may need.

Additionally, the last lines include a callback function, which is called after all instances have been created with a dictionary of all the instances, keyed by their URI. In this callback, a node is automatically added for all plugin instances, so the graph is populated immediately when the user loads the webpage.

All the specified plugins come with Celestrium, except for DependencyProvider. This is the plugin which is specific to this data set and is responsible for dynamically populating the GraphModel with information as needed. This involves defining two functions. The first is `getLinks`, which takes in a node and an array of nodes and returns the links from the single node to each of the other nodes. In most applications, this would involve querying a server and using the response to create the links. In this case, the dependencies are available already, so we simply check if either node depends on the other, and adds a link accordingly to the output.

```
getLinks: (node, nodes, callback) ->
  needs = (a, b) ->
    A = celestrium.defs[a.text]
    output =  A.needs? and b.text in _.values(A.needs)
    return output
  callback _.map(nodes, (otherNode) ->
    if needs(node, otherNode)
      return {strength: 0.8, direction: "forward"}
    if needs(otherNode, node)
      return {strength: 0.8, direction: "backward"}
    return null
  )
```

In Alar, this is where the the client would contact the server to find the polynomial in the rank of the SVD selected by the client which represented the link's strength. This illustrates that Celestrium is flexible enough to handle dynamically changing link strengths.

Next, `getLinkedNodes` provides an array of nodes which are linked to any of the given nodes. Again, since that information is already available, it can simply be calculated immediately and used.

```
getLinkedNodes: (nodes, callback) ->
  callback _.chain(nodes)
    .map((node) ->
      needs = celestrium.defs[node.text].needs
      needs ?= {}
      return _.values needs
    )
    .flatten()
    .map((text) -> {text: text})
    .value()
```

And that's it! This is all the implementation that is needed to render the graph of plugin dependencies which can be dynamically explored, as see in Figure 4-1.

4.4 Justification of the Data Provider Interface

Choosing what interface the user which would have to implement to provide data to the graph was a huge design decision. Ultimately, this was chosen because it is simple and separates the creation of nodes and links.

An alternative was implemented where a single function was used to populate the graph, however it then imposed a very confusing specification, although it did provide an opportunity for more efficient searching in Alar. Ultimately, the chosen interface was deemed more appropriate because of its simpler specification.

Chapter 5

Evaluation

Now that the conceptual and implemented contributions of this thesis have been presented, this chapter presents several forms of evaluation of this thesis's work. First, Section 5.1 describes a user study of Alar in which its use in finding erroneous assertions in a KB is tested against Microsoft Excel. It supports the hypothesis that Alar's interface is usable enough that new users can understand the interface and effectively perform the necessary operations to determine erroneous input data. Then, Section 5.2 describes an empirical analysis of this thesis's novel metric for assertion relatedness, finding it to be a good proxy for determining which assertions caused other assertions to be inferred.

5.1 Usability Testing of Alar

This section describes the specific experimental design used for testing Alar. It is first summarized here and the rest of this section explains it in full detail.

In summary, users who were comfortable using computers but haven't had experience with common sense reasoning were recruited. Each user was given a brief introduction to how common sense reasoning works. Then, each user was presented with an assertion that was incorrectly inferred to be true, and the user was asked to determine which of the original assertions may have caused this. There were asked to do this twice, once using a

Microsoft Excel spreadsheet of all the original assertions, once using Alar. After each of the two exercises, the user was asked questions about the interface on a Likert scale. Then the user was asked specific questions about Alar, again using questions based on a Likert scale. Lastly, they were asked questions to compare their experience using the Excel with their experience using Alar.

The rest of this section more thoroughly describes this experimental design and the results.

5.1.1 Briefly Explaining Common Sense Reasoning

For this user study, users were recruited who were comfortable using a computer but did not have background in common sense reasoning. The tasks they are required to perform as part of this study, however, require a basic understanding of the premise.

For consistency, the following text was used to explain common sense reasoning to each user.

> We have an inference process by which a computer is given a list of simple facts which are assumed to be true. It then generates new facts which it believes to be true based on the original facts.
>
> Unfortunately, if incorrect facts are fed into the inference process, this may cause it to generate incorrect facts as well.
>
> Therefore, if we find a fact which the inference process has generated to be false, we would like to identify which of the original facts may have lead to this happening. This is what we will be asking you to do.
>
> Specifically, we will give you an incorrect fact which the inference process has generated. We will then ask you to see if you can find any incorrect facts in the original list which may have caused this to be true. We will do this two different times, providing you with a different tool for answering this question each time.
>
> This is an evaluation of our software, not you. If you don't understand something or don't know what to do, that's actually very helpful for us. We would really appreciate you always speaking your mind and verbalizing your thought process if possible, both for positive and negative aspects of the experience.

5.1.2 Using a Spreadsheet

The first tool given to the user is a spreadsheet of all 225,202 facts. They are then given ten minutes to, given an incorrectly inferred fact, determine which facts in the original list may have potentially caused this fact to be inferred.

A spreadsheet is used because no other interfaces exist which leverage the AnalogySpace framework for inference, so if one were to be debugging a KB in practice, the KB would most likely be in the form of a spreadsheet. In short, it is the next best alternative to manipulating a KB to the author's knowledge. The following prompt was then presented to the user.

> Now, given these original facts, the inference process has inferred "car IsA street," which is not quite true. Using this spreadsheet, please identify some incorrect, original facts which may have lead to this incorrect fact to be generated. You will have ten minutes. Remember, please speak aloud your thought process as best you can and remember, we are not testing you, we are testing our interface.

5.1.3 Using Alar

Then, the user is given Alar instead of the spreadsheet and asked to perform the same task, only with a different incorrect assertion.

Much like needing to explain common sense reasoning, the user is given a quick introduction to Alar's interface. Here is the text that was used:

> Here is an alternative interface which visualizes the same information as the spreadsheet, just in a graph form.
>
> Nodes represent facts. Black nodes are original facts which were fed into the inference process. Green nodes are facts which the inference process generated.
>
> Links indicate how related any two facts are. Thick links are stronger, indicating the facts are more related. If two nodes have no links between them, they could be loosely related, but not enough to merit a link. You can include more or less links by adjusting this slider.
>
> You can space the nodes more tightly or farther apart using this slider.
>
> Lastly, you can search for any fact by typing it into this search bar and hitting enter. This will automatically show the graph centered around this fact and will

include its closely related facts. This includes original facts and newly generated facts.

For example, I can type in "bread IsA food" and this is what happens. (type in "bread IsA food" and type enter)

Once the interface has been explained, the user is then given the actual task.

Now, given these original facts, the inference processed has inferred "cup IsA drink" which is not quite true. Using this interface, please identify some incorrect, original facts which may have lead to this incorrect fact to be generated. You will have ten minutes. Remember, please speak aloud your thought process as best you can and remember, we are not testing you, we are testing our interface.

5.1.4 User Questions

Immediately after performing each task, the user was asked the following Likert scale questions.

Please state to what degree you agree with each statement

- This interface was easy to use.
- I was able to find the facts I needed to know.
- I was confident the facts I suggested led to the given incorrect fact being generated.
- I enjoyed using the interface.

The options are:

1. strongly disagree
2. disagree
3. neither agree nor disagree
4. agree
5. strongly agree

Additionally, after the task using Alar is finished and the above questions are asked, the following questions were asked which were specific to Alar. This interface was easy to use.

- I understood what the interface did.
- The visual representation was clear.
- The operations of the interface were helpful.

Finally, the following questions were asked in an effort to get feedback about comparing using a spreadsheet to using Alar.

If you had to do this as part of your daily work, which would you choose?

- spreadsheet
- website

These procedures were done for five different users and the results are analyzed next.

5.1.5 Analysis of Results

We begin with Figure 5-1, which shows the average level of agreement from all users for each of the questions they were asked for both interfaces. For all questions it was found that the level of agreement was greater when using Alar, $C^2 = (1, N = 5) \geq 6.75,\ p \leq 0.01$ for all tests.

Additionally, here are the statements to which users had to give their agreement that were particular to Alar and the average agreement:

- "I understood what the interface did." had an average of 4.0/5, equivalent to "Agree."
- "The visual representation was clear." had an average of 4.5, which is between "Agree" and "Strongly Agree".
- "The operations of the interface were helpful." had an average of 4.75, which again is between "Agree" and "Strongly Agree".

As a last question, users were asked, "If you had to do this as part of your daily work, which would you choose?" and all five users chose Alar.

Based on these results, we conclude that Alar is an improvement over standard data manipulation interfaces such as a spreadsheet to explore potentially problematic assertions in a knowledgebase.

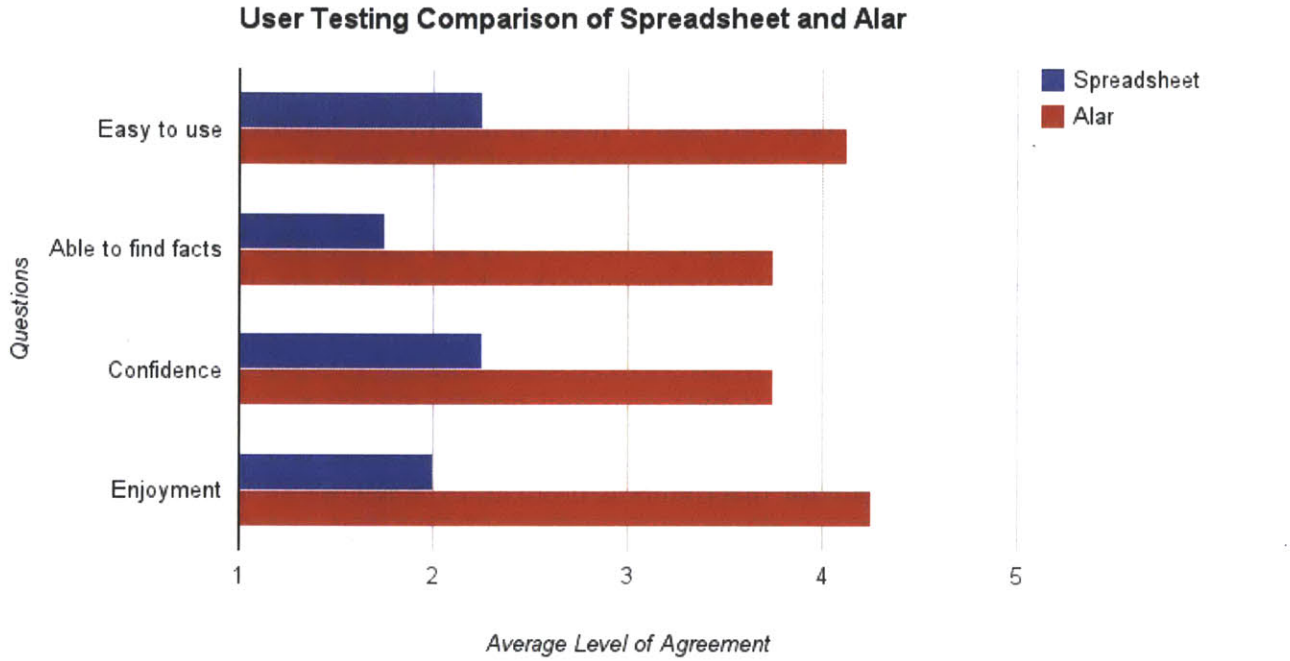

Figure 5-1: Comparing user responses reveals Alar was an improvement in all areas which were discussed with the users.

5.2 Evaluation of Assertion Relatedness

While people are certainly the end users for Alar's interface, further work was done to separately test the mathematical constructs underlying it. In particular, the novel technique of representing assertions as vectors was investigated.

In the user testing, users would often search for the given assertion and from the assertions which were populated on the graph, they would choose some which made sense to them. In this setting, assertion relatedness would hopefully be a good proxy for identifying assertions which led to the central assertion being inferred. As a followup to the user testing, this notion was examined in more detail in the following way.

For the same assertion as used in the user testing, "cup IsA drink," the forty most related assertions were calculated just as they would be for the interface. Then, starting with the most related assertion, it would remove the assertion from the KB and recalculate

the inferred truth of "cup IsA drink." It was then calculated how much the inferred truth of "cup IsA drink" changed, as a fraction of its original truth.

For example, "something AtLocation cup" was found to be the most related assertion. So its entries in the original assertion matrix were set to 0, the inference process redone, and the truth of "cup IsA drink" was found again and compared to the value found using the original, full knowledgebase.

This was repeated for the forty most related assertions. At each step the previously zeroed entries in the assertion matrix were left to be zero, so the results indicate the impact of removing all the first n related assertions, not just that particular one.

This isn't a very viable option for a user interface because an SVD of a matrix of the scale used takes at least several seconds, and an SVD must be computed for each of the top assertions we are considering.

Figure 5-2 is illustrates how much these forty assertions contributed to the inference of "cup IsA drink."

In summary, the forty most related assertions to "cup IsA drink" accounted for almost 80% of its value. To give some meaning to this number, there are 22,144 different concepts and 116,760 different features in this knowledgebase. Their cross product reveals all the 2563389440 possible assertions this KB can consider, each of which could potentially contribute to the inference of this assertion. This technique had identified forty assertions that accounted for 78% of the inference.

That is, it identified the $40/2563389440 = 0.00000156\%$ of assertions which accounted for 78% of the inference. I feel this is grounds to claim that the relatedness metric defined in this thesis results in a good proxy by which users can identify assertions which may have caused other assertions to be inferred.

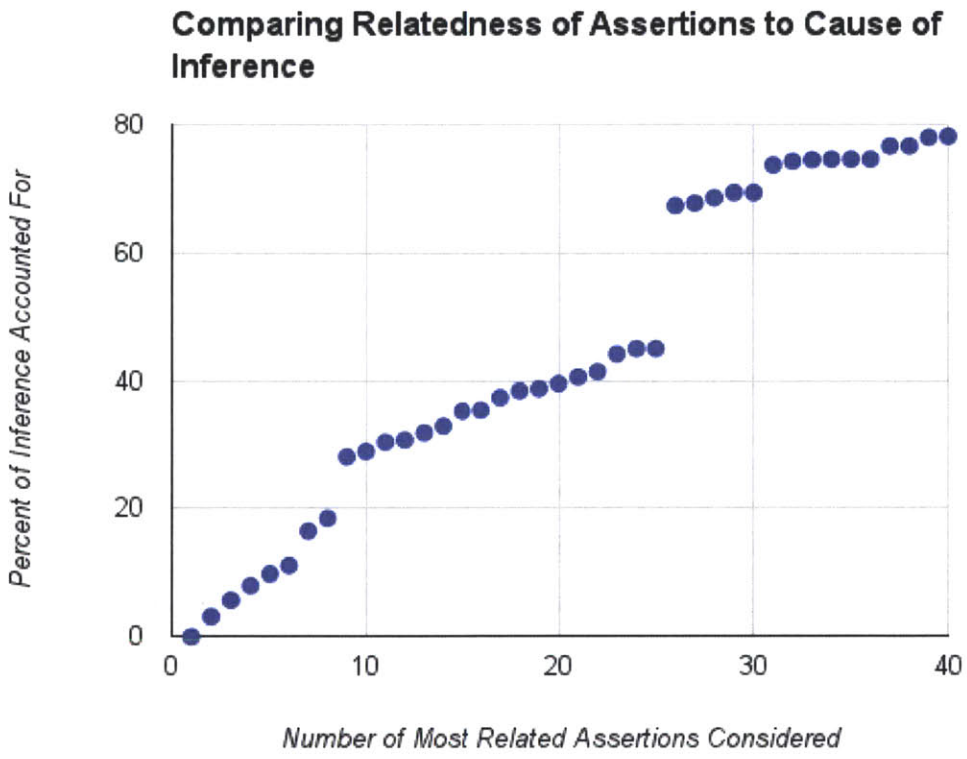

Figure 5-2: This chart illustrates the relationship between relatedness and cause of inference between assertions for the assertion "cup IsA drink", which was used in user testing. It reveals, for the x most related assertions, what fraction of "cup IsA drink's" inferred truth value came from them.

Chapter 6

Related Work

There has been surprisingly little work in visualizing inference in AI. While visualization has made enormous contributions to many scientific fields, AI has been slow to realize the potential of visualization.

In AI approaches that use logical inference, there has been work on visualizing the proof process itself. Most visualizations take the form of proof trees of the derivation of an assertion. Perhaps the best known of these is the Transparent Prolog Machine [8], a debugging tool for Prolog programs that visualizes the resolution proof procedure. It is especially useful for understanding the backtracking behavior of the Prolog theorem prover. But very little work in logical inference visualization treats the inference of more than one assertion at a time if they are not in the same inference chain. Few prior works attempt to visualize a semantic space of related assertions.

Of course, there have been many visualizations of static structures in knowledge bases, such as visualizations of ontologies. [9] surveys such systems. A good example is Protege [12], a comprehensive development environment for Semantic Web ontologies that provides a gallery of different visualizations, including node-and-arc diagrams, tag clouds, treemaps, and other popular visualization techniques for hierarchical data structures.

In statistical approaches there has been much work on visualizing the results of inference on large-scale data sets, or of visualizing large-scale data structures that control the inference.

Visualizing the result of an inference process on a large data set is usually specific to the kind of data involved. Image processing techniques that use machine learning visualize the results of their inference as images output by the inference. If the image looks good (by whatever standard), the inference is correct. But visualizing the output data of an inference process sometimes doesn't deliver much insight into "what the machine is thinking". As in the case of logical inference, the visualizations are mostly concerned about a single example at a time. However, [1] do organize sets of examples into visualization spaces that facilitates larger-scale understanding of a machine learning algorithm. Amershi? ??s work is one of the few to stress the importance of visualization and human feedback in AI inference algorithms.

Visualizing the data structures that control the inference is almost always specific to the kind of inference involved, and may not be immediately understandable to end-users who do not know the inference technique in detail. [11] visualizes neural networks; [5] and [3] visualize Bayesian networks; [15] show a confusion matrix for ensemble machine learning methods.

Previous work in visualization specifically for the ConceptNet knowledge base and AnalogySpace inference appears in [13]. AnalogySpace inference is used to compute "semantic dimensions" that each represent a spectrum of distinctions between concepts, such as "good vs. bad" or "easy vs. hard". Each concept is a point in this multi-dimensional space, and it can be explored through a standard 3D fly-through visualization. It doesn't directly visualize assertions.

Lastly, in the area of visualizing artificial intelligence processes for educational purposes, [2] presents an interactive, "Belief and Decision" Network Topology tool for creating and solving decisions trees. What distinguishes Alar from this visualization apart from the underlying process which it represents, is that the topology of the graph must be laid out by hand by the user and is not automatically organized or clustered in the same way as Alar.

Chapter 7

Conclusion

This chapter concludes this thesis by summarizing its contributions, suggesting ideas for future research in this area and ending with some final thoughts.

7.1 Summary of Contributions

This thesis's contributions started with the novel abstraction of semantic entity vectors within the AnalogySpace framework for inference. From there, it used the linearly separable model of an SVD to form vector representations of not only concepts, but features and assertions as well, allowing them to be compared for relatedness just like concepts. Additionally, semantic entity vectors allow efficient access to the results of the inference process as if the SVD had been computed for any lower number of singular values without having to actually recompute an SVD.

Next, this thesis presents its working implementation of these advancements, Alar, a web interface for dynamically exploring the results of inference over a knowledgebase. Alar's interface visualizes semantic entities as a graph, clustered by the relatedness of its nodes. Alar can also update its interface in realtime to visualize the results of inference for lower rank SVDs. A major component of Alar's implementation was Celestrium, a CoffeeScript library for visualizing large, graph based data sets and is an independent contribution of this

thesis as well.

Lastly, this thesis presents an evaluation of the above contributions through a usability study of Alar to find erroneous assertions in a Knowledgebase as well as a computational analysis of the effectiveness of this thesis's novel assertion relatedness metric.

7.2 Ideas for Future Work

Moving forward, there are several avenues which could yield exciting improvements in this space.

More generative than evaluative user testing may reveal improvements or new features that could benefit the interface. For example, much like the link strength distribution, when assertions are being shown, it might make sense to show the assertion strength distribution as well. The usefulness of this improvement could only be judged through user testing, though.

While the results of finding related assertions was found to be very successful, the algorithm itself uses a bit of a heuristic in that it only considers assertions which share at least a concept or feature. However, assertions could be related that do not have either of these in common. If the algorithm could be improved so that it truly did find the most related assertions, that would be a more authentic solution. It is currently not done because it would require calculating trillions of vector similarities, which isn't feasible for a realtime interface.

Lastly, the interface is meant to organically cluster semantic entities based on their relatedness. How well does it actually portray its underlying data? An evaluation of the interface's success in this sense would be very interesting. Furthermore, if this "authenticity" of the interface could be communicated to the user as part of the interface itself, in real time, it would just be that much more useful knowledge to provide to the user.

7.3 Final Thoughts

Common sense reasoning is not a widely used technique for inference. I hypothesize this is in part due to the lack of infrastructure and usable products surrounding it. It is my hope that the contributions of this thesis alleviate some of these problems and enable more end users and researchers to explore this exciting space.

Appendix A

Tables

Table A.1: List of Relations Used in this Thesis

IsA	What kind of thing is it?
HasA	What does it possess?
PartOf	What is it part of?
UsedFor	What do you use it for?
AtLocation	Where would you find it?
CapableOf	What can it do?
MadeOf	What is it made of?
CreatedBy	How do you bring it into existence?
HasSubevent	What do you do to accomplish it?
HasFirstSubevent	What do you do first to accomplish it?
HasLastSubevent	What do you do last to accomplish it?
HasPrerequisite	What do you need to do first?
MotivatedByGoal	Why would you do it?
Causes	What does it make happen?
Desires	What does it want?
CausesDesire	What does it make you want to do?
HasProperty	What properties does it have?
ReceivesAction	What can you do to it?
DefinedAs	How do you define it?
SymbolOf	What does it represent?
LocatedNear	What is it typically near?
ObstructedBy	What would prevent it from happening?
ConceptuallyRelatedTo	What is related to it in an unknown way?
InheritsFrom	(not stored, but used in some applications)

Table A.2: Core Celestrium Plugins

Celestrium	Defines plugin architecture.
DataProvider	Abstract framework for providing data set.
DependencyProvider	Provides data set as Celestrium plugin dependency graph.
ForceSliders	Provides slider to adjust parameters of graph visualization.
GraphModel	Maintains state of nodes and links.
GraphView	Renders actual graph of nodes and links.
KeyListener	Allows plugins to listen for hotkeys.
LinkDistribution	Renders filterable distribution of link strengths.
NodeSelection	Creates ability to select nodes.
SelectionLayer	Creates ability to select nodes in an area.
Sliders	Allows other plugins to add uniformly formatted sliders.
StaticDataProvider	Provides data if given static list of nodes and links.
Stats	Allows other plugins to add uniformly formatted labeled, statistics.

Bibliography

[1] Amershi, S., Fogarty, J., Kapoor, A., and Tan, D.(2011) Effective End-User Interaction with Machine Learning. In Proceedings of the AAAI Conference on Artificial Intelligence (AAAI 2011), Nectar Track, pp. 1529-1532.

[2] Amershi S. Tools for Learning Artificial Intelligence, AIspace. http://www.aispace.org/index.shtml, May 2014.

[3] Becker, B., Kohavi, R., and Sommerfield, D. Visualizing the simple Bayesian classifier. In Fayyad, U, Grinstein, G. and Wierse A. (Eds.) Information Visualization in Data Mining and Knowledge Discovery, (2001), 237-249

[4] Bostock, M. Data Driven Documents, D3JS. http://www.d3js.org, February 2014.

[5] Cossalter, Michelle, Ole J. Mengshoel, Ted Selker, Visualizing and Understanding Large-Scale Bayesian Networks, AAAI Workshop on Scalable Integration of Analytics and Visualization, San Francisco, California, USA, 2011.

[6] Havasi, Catherine, James Pustejovsky, Robert Speer, and Henry Lieberman, Digital Intuition: Applying Common Sense Using Dimensionality Reduction, IEEE Intelligent Systems 24(4), Special Issue on Human-Level Intelligence, pp. 24-35, July 2009.

[7] Havasi, Catherine, Robert Speer and James Pustejovsky, Coarse Word-Sense Disambiguation Using Common Sense. AAAI 2010 Symposium on Common Sense Knowledge.

[8] Eisenstadt, Marc, Mike Brayshaw, Jocelyn Paine, The Transparent Prolog Machine: Visualizing Logic Problems, Springer, 1991

[9] Katifori, A., Halatsis, C., Lepouras, G., Vassilakis, C., and Giannopoulou, E. 2007. Ontology visualization methodsA survey. ACM Comput. Surv. 39 , 4, Article 10 (October 2007).

[10] Krasner, G., Pope, S. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming archive Volume 1 Issue 3, Aug./Sept. 1988 Pages 26 - 49

[11] Olden, J.D., Jackson, D.A. 2002. Illuminating the black-box: a randomization approach for understanding variable contributions in artificial neural networks. Ecological Modelling. 154:135-150.

[12] Protege Project, http://www.protege.stanford.edu. Protege's visualization facilites are described in http://protegewiki.stanford.edu/wiki/Visualization, accessed February 2014.

[13] Speer, Robert, Catherine Havasi and Henry Lieberman, AnalogySpace: Reducing the Dimensionality of Commonsense Knowledge, Conference of the Association for the Advancement of Artificial Intelligence (AAAI-08), Chicago, July 2008.

[14] Speer, Robert, Catherine Havasi, Nichole Treadway, and Henry Lieberman, Finding Your Way in a Multi-dimensional Semantic Space with Luminoso, ACM International Conference on Intelligent User Interfaces (IUI), Hong Kong, China, February 2010, pp. 385-388.

[15] Talbot, J., Lee, B., Kapoor, A., & Tan, D. S. EnsembleMatrix: interactive visualization to support machine learning with multiple classifiers. In Proceedings of the 27th international conference on Human factors in computing systems (pp. 1283-1292), 2009.