Narratarium: Real-Time Context-Based Sound and Color Extraction from Text

ARCHIVES

by

Timothy Peng

S.B., Course VI M.I.T., 2012

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

June 2013

© Massachusetts Institute of Technology 2013. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 24, 2013

Certified by ....
Catherine Havasi
Research Scientist
Thesis Supervisor

Accepted by ....
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Narratarium: Real-Time Context-Based Sound and Color Extraction from Text

by

Timothy Peng

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Narratarium is a system that uses English text or voice input, provided either real-time or off-line, to generate context-specific colors and sound effects. It accomplishes this by employing a variety of machine learning approaches, including commonsense reasoning and natural language processing. It can be highly customized to prioritize different performance metrics, most importantly accuracy and latency, and can be used with any tagged sound corpus. The final product allows users to tell a story in an immersive environment that augments the story-telling experience with thematic colors and background sounds. In this thesis, we present the back-end logic that generates best guesses for contextual colors and sound using text input. We evaluate the performance of these algorithms under different configurations, and demonstrate that performance is acceptable for realistic user scenarios. We also discuss Narratarium's overall design.

Thesis Supervisor: Catherine Havasi
Title: Research Scientist

Acknowledgments

I thank Catherine Havasi, my thesis supervisor. She provided me with insight, guidance, and support, without which this thesis and the product it presents would not exist.

I thank Rob Speer. He gracefully and patiently helped me become familiar with the important tools upon which this thesis is built. His work and support was invaluable from the very beginning of the thesis.

Finally, I thank my parents, Ching and Jo, and my sister, Tina. Their continuous and unquestionable support through the last twenty-two years has provided me with an environment where I could learn, grow, and succeed, and it is to them that I owe all my accomplishments to date.

Contents

List of Figures

Chapter 1

Introduction

Narratarium is a consumer-facing device, comprising a projector, speaker, and computer, that uses machine learning techniques to intelligently augment real-life storytelling with colors and sounds. It offers two immersive user experiences. In the first experience ("freestyle" mode), the user (storyteller) either reads text aloud or enters it into Narratarium using a keyboard. Narratarium displays the words using the projector, then processes the input in real-time and enhances the storytelling environment by either changing the ambient color of the room via projector or playing a sound effect via speaker. In the second experience ("story" mode), the user selects a pre-recorded story that is saved on the computer, rather than speaking or typing. After a story is selected, the experience proceeds as if the user were entering the story using a keyboard in freestyle mode. Here is an example of a potential freestyle mode user scenario (for this example, we assume Narratarium is set up in a room with white walls):

1. The user selects freestyle mode.

2. The user speaks, "The giant tree was ablaze with the orange, red, and yellow leaves that were beginning to make their descent to the ground..."

3. The projector displays the story, word by word as the words are spoken, on the walls of the room (*i.e.* "The" is displayed, followed briefly by "giant," then "tree," etc.).

4. When the word “tree” is displayed, the room’s walls turn green.

5. When the word “ablaze” is displayed, the user hears wood burning, and the room’s walls turn red.

6. When the word “orange” is displayed, the room’s walls turn orange.

7. When the word “red” is displayed, the room’s walls turn red.

8. When the word “yellow” is displayed, the room’s walls turn yellow.

9. The user experience proceeds in a similar manner until the story is finished.

The story mode user scenario is nearly identical:

1. The user selects story mode.

2. The user selects the story, “The giant tree was ablaze with the orange, red, and yellow leaves that were beginning to make their descent to the ground...”

3. The projector displays the story, word by word, on the walls of the room (*i.e.* “The” is displayed, followed briefly by “giant,” then “tree,” etc.) as if a human is manually entering it into Narratarium.

4. The user experience is identical to the freestyle mode experience after this point.

At the highest level, the Narratarium system is composed of three parts, which together offer the user experiences described above:

- Hardware that supports all the necessary functionality (*e.g.* full-room projector, speaker, microphone, computer)

- Front end software that can receive user input, whether by voice or by text, and can project words, colors, and sounds gracefully.

- Back end software that can receive text input and return colors and sounds based on the input.

My thesis work focuses primarily on the third requirement – back end software that can receive text input and return colors and sounds based on the input. It also provides an interface through which the front and back ends can communicate.

In the user scenarios described earlier, color change and sound effects occurred selectively (*e.g.* a sound effect only occurred following the word "ablaze"). Narratarium only provides color changes or sound effects that will enhance, and not detract from, the user experience. The back end helps achieve this in the following manner:

- **It decides what color or sound effect to use based on the available context.** Some words are only associated with color or sound when interpreted in context. For example, the word "bark" is not associated with a sound when presented in context with "tree," but is associated with a sound when in context with "dog." Similarly, the word "leaves" is not generally associated with a color unless it is considered in the context of "tree" or "autumn." Context also allows for fine-grained refinement in the interpretation of words that are strongly associated with a particular sound – for example, "loud rain" versus "soft rain." The back end considers not only the most recently entered word, but its context, when it makes its decisions.

- **It decides what color or sound effect to use based on the available color and sound corpora**. If the user enters a word, such as "bassoon," which is strongly associated with a sound that is infrequently recorded, it is possible that a file containing the sound is not present in Narratarium's sound corpus. However, given the context of the word, other sound files which are available in the corpus, such as a recording of another woodwind instrument, may suffice. The back end produces best guesses for sounds and colors based both on the input provided by the user and on the available sound and color corpora.

- **It provides a measure of confidence in its guesses**. Certain words (*e.g.* "honesty") are not commonly associated with a color or a sound, while other words are commonly associated with a color (*e.g.* "red"), a sound (*e.g.* "creak"), or both (*e.g.* "fire"). Although best guesses for color changes and sound effects are generated for every input word (with the exception of stop words such as "the" and "was"), an additional metric is presented that can indicate whether the guesses should be used to change the ambient color or play a sound, or if the guesses should be ignored.

- **It makes best guesses for associated color and sound in real time.** In "freestyle" mode especially, **latency** and **responsiveness** are very important, so efficient algorithms are essential. Because the processing power of the device is unknown, the back end algorithms are highly configurable to ensure adequate performance, both in terms of time taken to process input and generate output, and in terms of accuracy of guesses.

This thesis describes the design and implementation of the back end. Dan Novy, in the MIT Media Lab's Object-Based Media group, is designing and implementing the front end.

Perhaps due to its limited practical application, color and sound extraction from text is an unexplored niche of natural language processing. There is little to no related work on the problem aside from research done at the MIT Media Lab. While ideologically similar branches of natural language processing research, such as contextual polarity analysis and document classification, are much more well-documented and well-explored, certain aspects of the color and sound extraction problem render the common machine learning approaches ineffective or inapplicable. For example, one might consider color extraction to be a multi-dimensional version of contextual polarity analysis. Rather than maintaining a scalar value corresponding to the magnitude of positive sentiment of a section of text, one might maintain a vector of values, each of which corresponds to the magnitude of "red" sentiment, "green" sentiment, and "blue" sentiment. But even if this were possible, the vector of color sentiment values would not provide nearly enough information to generate an accurate representative color for a section of text (for example, the phrase "brick wall" may be more "red" than "green" or "blue," but this information alone says nothing about what shade of red should be provided, or if green and blue color components should be present but muted). Meanwhile, sound extraction can be seen as a classification problem (there is no intuitive way to model sound extraction as a regression problem). However, there would be as many class labels as sound files in the sound corpus, and any sound corpus that can be claimed to adequately represent the entire human sound experience will surely contain thousands, if not hundreds of thousands or millions, of sound files. Common text classification approaches, such as Naive Bayes or EM, do not scale well enough with number of class labels to support real-time classification. Narratarium uses efficient and scalable algorithms to extract colors and sounds from words.

The rest of this thesis details the design, implementation, and evaluation of Narratarium's back end. Chapter 2 covers the front end work and pre-existing work related to the back end. Chapter 3 covers the design and implementation of the system. Chapter 4 presents an evaluation of the performance of the different implementations. Chapter 5 provides a conclusion and possibilities for future development.

Chapter 2

Related Work

As mentioned earlier, traditional machine learning algorithms (*e.g.* Naive Bayes and EM) used for natural language processing and computational linguistics problems, such as contextual polarity analysis and document classification, can be either inefficient or inapplicable to the problem of contextual color and sound extraction. Thus the problem of real-time contextual color and sound extraction calls for new, scalable approaches.

Narratarium is a consumer-facing product comprising much more than machine learning algorithms. For example, the back end is a combination of new implementation (which will be discussed in the next chapter), re-purposed libraries developed previously at the MIT Lab, and corpora from outside sources. This chapter discusses the front end component of Narratarium, as well as the many tools upon which the back end relies.

2.1 Front End

Narratarium's front end has a number of responsibilities:

- Take a word as text input and pass it to the back end.
- Take a word as voice input, translate it to text, and pass it to the back end.
- Take a word as text input and display it elegantly through the projector.

- Receive an RGB value from the back end and elegantly change the projected background color to the RGB value, if it has not been changed recently.
- Receive a sound file location from the back end and play the sound file at that location. If a sound effect is being played concurrently, either discard the new sound effect or gracefully transition from the old sound effect to the new sound effect.

Dan Novy is developing the front end as a C++ openFrameworks project. Voice input recognition and translation of voice input to text input is accomplished using the Carbon speech synthesis API. Text input is then passed to the back end using a C++ to Python interface.

Text display is straightforward when the projection surface is a flat surface and is parallel to the plane of the projector (*e.g.* a wall or a computer monitor). However, Narratarium offers an immersive, full-room experience, so text is displayed elegantly regardless of the geometrical nature of the projection surface (in the typical user scenario, the projection surface will comprise four walls and a ceiling). Narratarium accomplishes this by requiring the user to go through an initial set-up phase, in which the user selects a series of control points, corresponding to the corners of each wall; Narratarium then creates and applies a set of homographies that allows text to be displayed properly on the walls for which the user provides control points.

Color changes and sound changes are handled in a straightforward manner. When a new RGB value is received from the back end, the front end calculates the amount of time since the most recent color change; if this exceeds a certain value, the front end begins a smooth transition from the original color to the new RGB value. Similarly, when a new sound effect (more precisely, a URL to an online .mp3 file) is received from the back end, the front end considers whether a sound effect is currently being played; if so, then the front end discards the new sound effect. Otherwise, the front end plays the new sound effect if the corresponding certainty value that is passed along with the sound effect exceeds a certain threshold value. Thus, the front end maintains four state variables related to the back end: the current background RGB value, the time of the last background color change, the duration of the most recently played sound effect, and the time that the last sound effect was begun.

2.2 Open Mind Common Sense (OMCS)

Humans subconsciously and continuously build and augment their commonsense knowledge through everyday experiences. As a result, they assume that a certain amount of commonsense knowledge, or set of facts about objects and the relations between them, is shared by all humans. They almost always do not explicitly state these facts when they communicate with other humans. This commonsense knowledge which humans take for granted, such as "a dog is an animal" or "an alarm clock is used for waking up," plays an critical part in deriving meaning from conversation, and can only be learned through real-life experience or through direct enumeration and memorization. This poses a major problem when attempting to interpret conversation using a machine, as machines do not possess human experience. Thus a means of compensating for the resultant lack in basic commonsense knowledge is necessary.

The Open Mind Common Sense (OMCS)[7] project attempts to enumerate and record the extensive amount of commonsense knowledge that humans possess and use. It accomplishes this goal by turning to the general public and allowing ordinary people to submit facts, rules, stories, and descriptions about any facet of everyday life. OMCS then parses these facts, rules, stories, and descriptions, and maintains a large database of structured English commonsense sentences.

2.3 ConceptNet

ConceptNet, as described by its founders, is a "freely available commonsense knowledge base and natural-language-processing tool-kit which supports many practical-reasoning tasks over real-world documents"[3]. It processes commonsense data from the OMCS corpus[7], in the form of sentences like "people generally sleep at night," and creates a graph network of nodes and edges that represents objects and the commonsense relationships between them.

First, ConceptNet uses a suite of natural language processing extraction rules to transform each English sentence in the OMCS corpus into a binary-relation assertion(a binary-relation assertion, as the name suggests, is a commonsense assertion relating two variables, *e.g.* `PropertyOf`(lime,sour) or `IsA`(lime,fruit)). It then normalizes the word components in each binary-relation assertion by stripping its three terms (the two variables and the assertion itself) of semantically peripheral features such as tense and multiplicity (thereby acknowledging and accounting for the intuitive equivalence of assertions such as `IsA`(lime,fruit) and `Are`(limes,fruits)). It also applies a number of semantic and lexical generalizations to further prevent semantic- or lexical-based multiplicity of intuitively equivalent assertions. After accomplishing this, ConceptNet possesses a large set of normalized, intuitively unique binary-relation assertions; it uses this set of assertions to construct a graph network. Each node in the network corresponds to a unique term in a binary-relation assertion, and a directed edge with label `L` exists from node B to node A for each binary association `L`(A,B). Figure 2-1, provided by [3], depicts a particular subgraph of ConceptNet's association graph. A few of the assertions that were used to generate the subgraph include `UsedFor`(alarm clock, wake up), `LocationOf`(kitchen, in house), and `SubeventOf`(chew food, eat breakfast). Note that the an edge leaves the node corresponding to the second term in the assertion and enters the node corresponding to the first term in the assertion.

Each edge in the graph is also weighted with the frequency of its corresponding binary-relation assertion (including semantically and lexically equivalent assertions) in the original OMCS corpus. The weight of an edge connecting two terms can be interpreted as indicating how strongly the two terms are associated; an assertion between two words is "stronger," and the two words are more strongly associated, if many people submitted the assertion to OMCS than if fewer people submitted the assertion to OMCS.

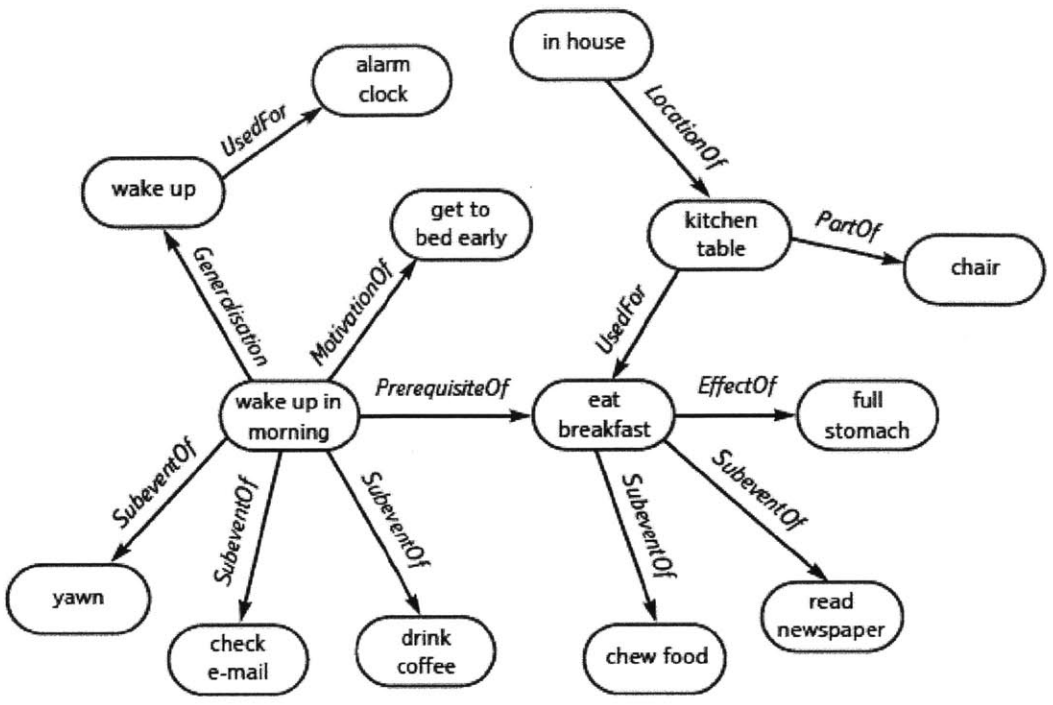


Figure 2-1: A Subgraph of ConceptNet's Assocation Graph

2.4 Divisi

ConceptNet's association graph contains an extremely large number of nodes and edges. The graph is populated using the structured English sentences in the OMCS corpus, for which little moderation of submission quality or legitimacy exists. As a result, weak, uncommon, or even false commonsense assertions are included in the corpus, and are transferred into the association graph.

Because OMCS draws its commonsense sentences from a very large number of users, Divisi makes the assumption that strong assertions (assertions that are more likely to be true) will have higher weights in the ConceptNet association graph than weak or faulty assumptions – that is, a true or commonly held assertion is likely to be submitted more often than a weak or false assertion.

Divisi[8] uses an adjacency matrix to represent ConceptNet's association graph. Because the graph contains an extremely large number of nodes, the adjacency matrix is of extremely high dimensionality, and operations involving the matrix can be extremely inefficient. However, the adjacency matrix is also very sparse. Divisi dramatically reduces the dimensionality of the adjacency matrix by pruning weakly connected nodes and the assertions about them (*i.e.* reducing the adverse effect of weak and faulty assumptions). It does this using singular value decomposition (SVD) [8]. Divisi uses a number of Python toolkits (including NumPy, PySparse, and SVDLIBC) to factor the adjacency matrix A into the product $U\Sigma V^T$ of the orthonormal matrices U and V^T and the diagonal matrix Σ. The diagonal matrix Σ has along its diagonal the many singular values of A. Divisi rejects the smallest singular values of Σ, and the rows and columns of U and V^T to which they correspond, thereby reducing the dimensionality of U, Σ, and V^T, and thus also reducing the dimensionality of their product. The result is a new adjacency matrix of much smaller dimension than the original adjacency matrix, which represents a pruned version of the original ConceptNet association graph.

This adjacency matrix dictates the existence of a commonsense assertion, and measures its strength/legitimacy, between two words. Divisi introduces a new feature called "activation spreading" [8] which measures the strength of the semantic relationship between any two words, regardless of whether a commonsense assertion exists between them, and independent of what the assertions actually are. It does this by turning the underlying directed association graph into an undirected association graph (applying a graceful transformation of directed edges to undirected edges), then generating a new square, symmetric adjacency matrix for the undirected association graph. It then performs a number of matrix operations on the new adjacency matrix [2] such that the final result is a square matrix *divisi* such that $0 \leq divisi[i,j] \leq 1$ and $divisi[i,j]$ estimates the semantic relatedness between words i and j.

It is important to emphasize that this measure of semantic relatedness is completely determined by the nature of the original OMCS corpus, and thus the semantic relatedness between two words provided by Divisi is truly the semantic relatedness between two words according to the humans who submitted commonsense knowledge to OMCS. However, the OMCS corpus samples commonsense knowledge from more than 16,000 unique contributors [7], so it is likely that *divisi*'s semantic relatedness

```
>>> divisi.row_named('red').top_items(5)
[(u'red', 0.70358479022979736), (u'color', 0.68834376335144043), (u'colour', 0.6
6469812393188477), (u'blue', 0.65805768966674805), (u'orange', 0.629241645336151
12)]
>>> divisi.row_named('bassoon').top_items(5)
[(u'band', 0.54526448249816895), (u'instrument', 0.54265880584716797), (u'musica
l instrument', 0.53656864166259766), (u'orchestra', 0.53629171848297119), (u'mak
e music', 0.53453058004379272)]
>>> divisi.row_named('limousine').top_items(5)
[(u'car', 0.16289833188056946), (u'automobile', 0.14575172960758209), (u'vehicle
', 0.14069372415542603), (u'drive', 0.13873940706253052), (u'motor', 0.134099721
90856934)]
```

Figure 2-2: Example Relatedness Values using Divisi

values are reasonably accurate.

Figure 2-2 presents the five most semantically related words for each of the words "red," "bassoon," and "limousine" using the matrix *divisi*. *divisi* contains rows and columns for 66,375 English words, but we include only the top five for the sake of brevity.

An interesting characteristic of the semantic relation matrix *divisi* is that $divisi[i, j] > divisi[i, i]$ for certain words i and j, implying that certain words are less semantically related to themselves than to other words. For example, $divisi['bassoon','bassoon'] = 0.423$ while $divisi['bassoon','band'] \approx 0.543$. This seemingly counterintuitive phenomenon is a side effect of the dimensionality reduction and matrix operations performed upon the original association matrix. The back end algorithms use the semantic relation matrix *divisi* and compensate for this side effect.

2.5 XKCD Color Corpus

Humans learn color-word associations through their everyday experiences, just as they do commonsense knowledge. For example, humans associate the word "apple" with red and green tones either because the apples they see are generally red or green, or because someone told them so.

Obviously the latter method of learning color-word associations is easier for a machine to replicate, and so the need arises for a color association corpus which maps English words to colors that are commonly associated with them.

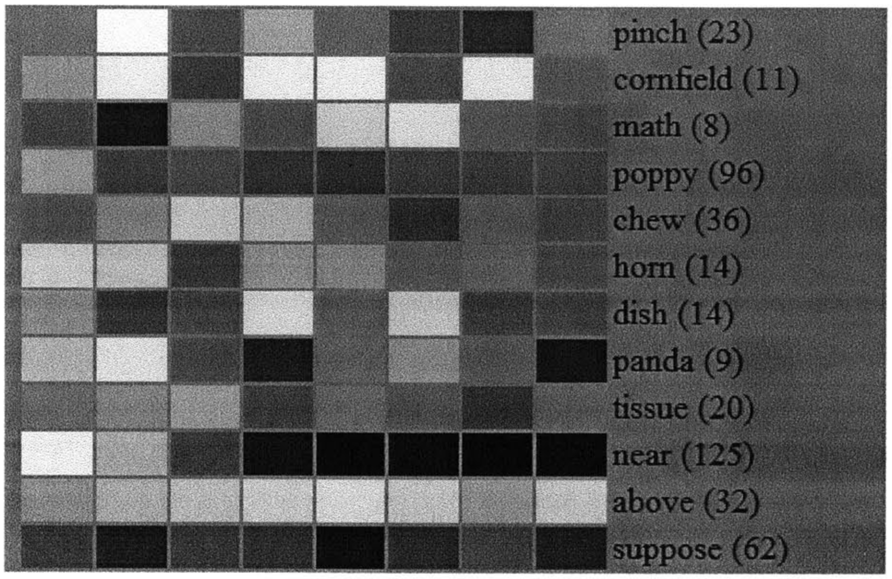


Figure 2-3: Example XKCD Color Associations

Randall Munroe (of XKCD fame) completed this task previously[4], and provides a corpus mapping English words to sets of commonly associated RGB values for those words.

Figure 2-3 presents a graphical representation of a subset of the XKCD color association corpus. Each of the words in the figure is followed by the number of unique RGB color values that people have associated with the word and preceded by the swatches for eight RGB values randomly sampled from these values. For example, the word "above" was associated with 32 different colors; it just so happens that the eight random samples from these 32 colors are all different shades of green. The corpus does not keep track of the frequency of RGB value - word associations.

2.6 Colorizer

Colorizer[2] is an application that "hypothesizes color values that represent a given word or sentence." It uses the XKCD color corpus[4], in conjunction with two other color corpora, to generate an extensive corpus mapping 11,009 English words to sets of RGB values commonly associated to those words.

Colorizer works by maintaining a set of weighted RGB values corresponding to the colors associated with any words previously entered (this set is initially empty). When presented with a new word W, Colorizer performs a few steps to update the set:

1. Decreases the weights already in the set by a constant factor (thus imposing a sense of "decay" so that more recently entered words receive priority in color associativity over less recently entered words).

2. Checks if W is in the corpus; if it is, adds the colors to which it maps to the set with weight 1.0.

3. Finds the k (a configurable integer with default value five) most closely semantically related words using Divisi's semantic relatedness matrix *divisi*. Then for each of these words W_i, checks if W_i is in the corpus; if it is, adds the colors to which W_i maps to the set with weight equal to $divisi[W, W_i]$.

4. Re-weights each of the colors in the set as a function of its current weights as well as its Euclidean distance, in the LAB color space, from the other colors in the set.

5. Performs a single transferable voting election using the colors in the set and their weights to determine the "best" hypothetical associated color valuess.

Colorizer was developed by MIT Media Lab researchers Catherine Havasi, my thesis supervisor, and Rob Speer. The source code is mostly open source. Further implementation was performed to re-purpose the code in conformity to Narratarium's design.

2.7 Freesound Sound Corpus

For the purpose of this thesis, we constructed a sound corpus using a database dump of Freesound's online sound database[1]. It contains metadata for 72,999 unique sound files in the database. This will be discussed to greater depth in next chapter.

Chapter 3

Design and Implementation

This chapter discusses the back end's requirements, design, and implementation.

3.1 Requirements

Narratarium's back end is responsible for receiving, asynchronously, a series of words, and for each word, hypothesizing a RGB color value and a sound file (drawn from the Freesound corpus [1]). Furthermore, there are several requirements of the hypotheses which can be used to measure Narratarium's success. The hypotheses must be:

- **Context aware**. When the ith word W_i is received, the back end uses words W_{i-1}, W_{i-2},..., W_{i-j} (for some finite j) to provide a more contextually accurate hypothesis for W_i's color and sound values.

- **Accompanied with a measure of certainty/validity**. Although a color and sound guess will be generated for almost every English non-stopword (the exception being words which are not in Divisi's relatedness matrix *divisi* or any of the color or sound corpora), a color change or sound effect need not be played after every word (indeed, this would detract from the user experience). For each input word, the back end provides a number in addition to its hypotheses. This number indicates the back end's confidence in the guess, as well as how strongly the word appears to be associated with color or sound in the first place. The front end can use this metric in conjunction with its knowledge of the recency of the last color change or sound effect to determine whether or not to act.

- **Generated in real-time with low latency.** Like any other user experience, Narratarium must appear fast and responsive. Nielsen[5] states that a 0.1 second response time is sufficient to make the system seem as if it is reacting instantaneously. This would correspond to the time between when the user finishes entering a word and when a corresponding state change, if there is one, occurs. However, there are two important factors that allow for some leeway in this regard. First, Narratarium is used as part of a storytelling experience, where the listener is focused on the story rather than on the environment. In freestyle mode, the story is being told by one human to another, so the listener will be focusing on the storyteller and not on Narratarium's output. In story mode, where the listener is reading the story off the walls, the words are projected on the walls before being sent to the back end. Thus in both modes, the perceived delay between input and sound or color state change is marginalized by other parts of the user exerience. Second, when people tell stories, words that are in the same vicinity as each other may imply similar underlying sounds or colors (*e.g.* "red leaves," "blazing inferno,"). This will likely be the case when the story is describing something in detail (*i.e.* using many descriptive words for the same object or concept). It is situations like these that Narratarium especially aims to enhance with sound and color, and where higher latency is forgiveable. For example, if a sequence of words "$W_i, W_{i+1}, W_{i+2}, ..., W_j$" are all being used to describe a scene, it is acceptable if the corresponding color change occurs later than W_i, since $W_{i+1} ... W_j$ are also describing the same scene as W_i. Nevertheless, responsiveness and low latency are important parts of the user experience.

These three metrics for success can be easily measured and evaluated, and will be discussed later in this chapter and in Chapter 5.

A notable non-requirement of the feature is that for any given word W, the color hypothesis and sound effect hypothesis are the "most appropriate" hypotheses possible. There are two reasons for this.

First, there is no way to assign a number to a color or sound effect hypothesis which measures the hypothesis's "appropriateness" for a particular sequence of words, or to compare the "appropriateness" of two hypotheses on an absolute scale. The "appropriateness" of a hypothesis is decided by the end user, and is completely dependent on the end user's personal perceptions of color, sound, and the story being told. Assuming that the back end were to propose a certain RGB value for a set of words, it is very likely that any particular end user, if asked to choose an RGB color for the same set, would choose a color with a different RGB value, simply because there are 256^3 possible colors to choose from. Comparing the back end's proposed RGB value to a particular user's proposed RGB value would be meaningless because any other user would likely propose a different RGB value. Furthermore, even if all humans possessed the exact same color-word associations, there is no absolute and meaningful way to measure how much less appropriate the back end's proposed color value might be than the correct value.

Second, the success of Narratarium as an overall product is determined by whether it is able to enhance the user experience of the listener, not whether it is able to maximize the user experience of the listener (this is impossible). Thus, it suffices that a color or sound effect is "appropriate enough" for a set of words, and not that a color or sound effect is the "most appropriate" color, of all RGB colors, or sound effect, of all the sound effects in the corpus, for that set of words.

That the success of sound and color extraction from a text depends on the unspecified listener, and not the person who wrote the text, differentiates it from other NLP tasks. For example, text polarity analysis seeks to classify a set of words as having an overall positive, neutral, or negative sentiment. Its success is determined by whether its hypothesis matches the sentiment of the author of that set of words. Thus one can measure the success of a text polarity analysis algorithm on a specific piece of text by consulting its author to see if the hypothesized sentiment matches the

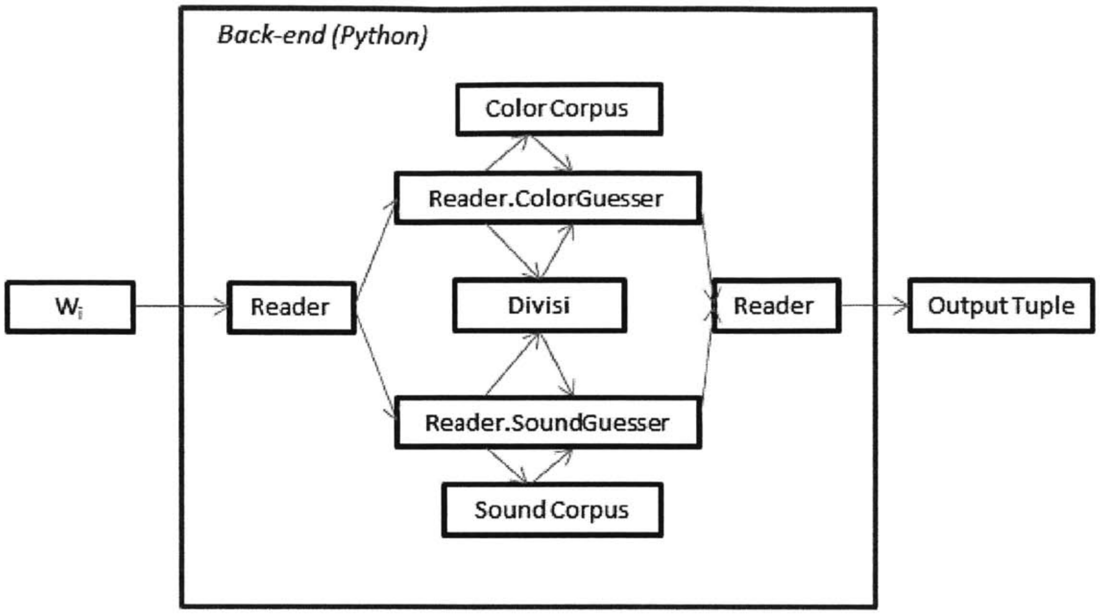


Figure 3-1: General Design of Back End

author's sentiment when he wrote the text. The same verification cannot be easily done for sound and color extraction.

3.2 Design

The design of the back end follows from the requirements enumerated in the previous section, and is depicted at the high level in Figure 3-1. The entire back end is contained inside a single Python object called `Reader`. When the front end is first initiated as a C++ OpenFrameworks project, it creates an embedded Python instance of `Reader`. Then when the user enters or speaks a word, the front end calls `Reader`'s update method, passing it the new word. `Reader` contains two subclasses, `ColorGuesser` and `SoundGuesser`, which function independently. `Reader` calls the update methods of both subclasses, passing them the new word, and the subclasses perform the necessary state updates and return their respective guesses. Each subclass utilizes its corresponding corpus, and Divisi's relatedness matrix *divisi*, in its computations. The guesses are then passed back to the front end.

3.2.1 Color Extraction vs. Sound Extraction

Although the tasks of color extraction and sound extraction are conceptually similar, the constraints imposed on guessing sound are very different than the constraints imposed on guessing color. Consequently, the approach used to extract color in [2] does not transfer gracefully to sound extraction.

Consider the full statement of each task:

- **Color extraction** seeks to provide a best guess for the color, chosen from the set of colors in a particular color corpus, associated with a word or set of words.

  When coming up for a best guess for the color to represent "fire," Colorizer[2] simply consults its color corpus to determine which colors have been associated with "fire," and chooses from these a particular color c such that the sum of the Euclidean distances, in the LAB color space, between each of the colors in the set and c is minimized. From a human's perspective, this is a meaningful and sensible optimization, since it is equivalent to choosing the color that is the least different from all of the other colors (or the most "neutral").

- **Sound extraction** seeks to provide a best guess at the sound file, chosen from the set of sound files in a particular sound corpus, such that the sound in the sound file is associated with a word or set of words.

  When coming up for a best guess for the sound file whose contents represent "fire," one can again consult a sound corpus to determine which files have been associated with "fire." If one were to continue using the strategy that Colorizer uses for color extraction, one would look at the waveform or bit representation for each file and choose the file whose waveform differs the least from the all of the rest. However, the file whose waveform is the least different from the waveforms of the other files is not the file whose underlying sound effect is the least different from the underlying sound effects of the other files. The distinction is that RGB values, which define colors, hold practical meaning for humans, while the waveform or bitwise content of a sound file, which defines the file, does not hold practical meaning for humans.

As a result, the algorithms for color extraction are different from the algorithms for sound extraction; consequently, `ColorGuesser` and `SoundGuesser` are completely modular and can function independently of each other.

3.3 ColorGuesser High Level Implementation

`ColorGuesser`'s implementation is a version of Colorizer's implementation[2] that has been modified in a few ways:

- Colorizer uses methods from a proprietary library, Luminoso[9]. Code was implemented that replicated the effects of these methods, without using the proprietary library.
- Small changes in implementation and in parameter settings were made to decrease computation time(latency).

Conceptually, however, `ColorGuesser` functions identically to Colorizer.

3.4 SoundGuesser High Level Implementation

The implementation of `SoundGuesser` can be conceptually split into three parts: the sound corpus, the metrics algorithms, and the guessing algorithms.

3.5 Sound Corpus Implementation

There was no publicly available sound association corpus which, similar to Colorizer's color corpus, mapped English words to sound files, so it was necessary to create one.

The bulk of Colorizer's color corpus was obtained through a global color survey by [4], in which volunteers were shown a set of colors and asked to provide a word or phrase related to each of the colors. Using the same approach to create a sound corpus was infeasible for a few reasons. First, it takes much more of the user's time to listen to a sound file than it takes to look at a color; in addition, it is much easier to generate sample colors than it is to generate meaningful sample sound files (sound files that play sounds humans are likely to encounter in everyday life). Thus, setting up and administering a poll would have been very tedious and painful for both the poll giver and the poll takers. Second, the global color survey sampled more than 222,500 users, a sample size that is far outside the scope of this thesis.

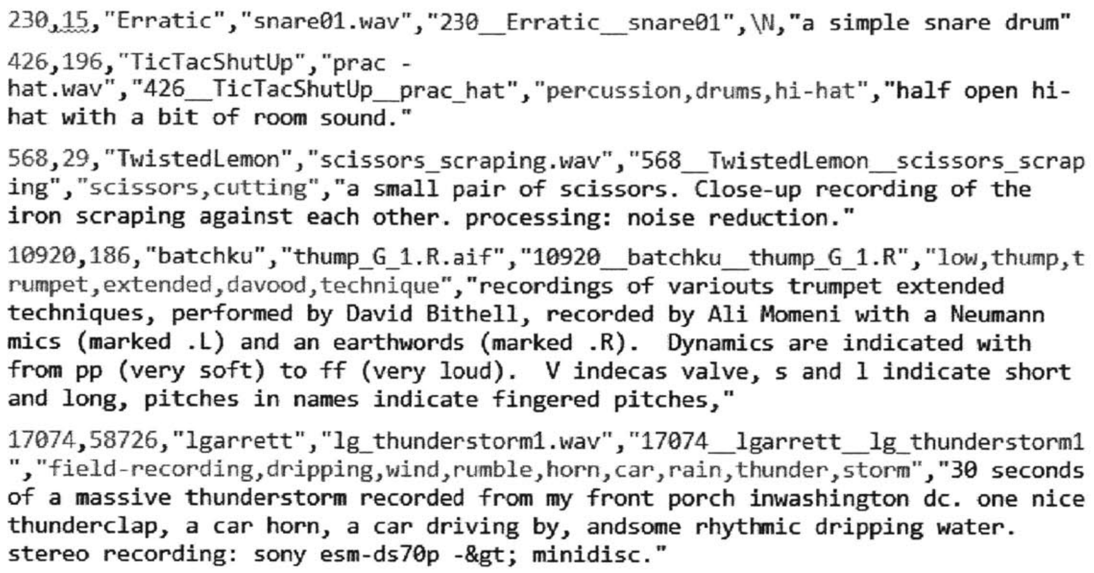
Instead, the color corpus was created using a database dump from Freesound, an open source online sound database[1]. Anyone can upload any sound file to Freesound's database. Uploaders are encouraged to tag their file with keywords and to provide a description string along with the upload. Figure 3-2 presents five samples of sound file entries from the Freesound database dump. Each entry in the dump corresponds to a unique sound file hosted on Freesound's website. The first two fields in each entry are tracking numbers assigned to the sound file for that entry, and can be used to generate a direct URL to the .mp3 of the sound file on Freesound's website. Together, they are unique for each sound file in the database and can be used as identification. The third, fourth, and fifth fields in each entry are the username of the uploader, the original name of the sound file for that entry, and the name of the file's corresponding preview file (generated by Freesound when the file is uploaded), respectively. The sixth field in each entry corresponds to a comma-separated set of keywords the uploader tagged the file with, or the value `\N` if the uploader did not tag the file with any keywords. Finally, the seventh field in each entry corresponds to the description of the file's content that the uploader provided during upload.

3.5.1 Sound Corpora Generation

Creating the corpus required parsing the Freesound dump and creating a few dictionaries:

- `TagsToIDs`, a dictionary which maps an English word to a set of file IDs of the sound files which have been tagged with that word as a keyword.

- `IDsToTags`, a dictionary which maps a file ID to the English words which the uploader tagged the file with (the reverse mapping of `TagsToIDs`).



```
230,15,"Erratic","snare01.wav","230__Erratic__snare01",\N,"a simple snare drum"
426,196,"TicTacShutUp","prac -
hat.wav","426__TicTacShutUp__prac_hat","percussion,drums,hi-hat","half open hi-
hat with a bit of room sound."
568,29,"TwistedLemon","scissors_scraping.wav","568__TwistedLemon__scissors_scrap
ing","scissors,cutting","a small pair of scissors. Close-up recording of the
iron scraping against each other. processing: noise reduction."
10920,186,"batchku","thump_G_1.R.aif","10920__batchku__thump_G_1.R","low,thump,t
rumpet,extended,davood,technique","recordings of varaiouts trumpet extended
techniques, performed by David Bithell, recorded by Ali Momeni with a Neumann
mics (marked .L) and an earthwords (marked .R).  Dynamics are indicated with
from pp (very soft) to ff (very loud).  V indecas valve, s and l indicate short
and long, pitches in names indicate fingered pitches,"
17074,58726,"lgarrett","lg_thunderstorm1.wav","17074__lgarrett__lg_thunderstorm1
","field-recording,dripping,wind,rumble,horn,car,rain,thunder,storm","30 seconds
of a massive thunderstorm recorded from my front porch inwashington dc. one nice
thunderclap, a car horn, a car driving by, andsome rhythmic dripping water.
stereo recording: sony esm-ds70p -&gt; minidisc."
```

Figure 3-2: Example Freesound Database Dump Records

- `IDsToMeta`, a dictionary which maps a file ID to all the metadata available for it from the dump.

There were a few ways do do this (described following this paragraph). In layman's terms, **Method 1** adds a file to the corpus only if it has been tagged with keywords, regardless of whether or not a description is provided for it. **Method 2** first completes **Method 1**, at which point `TagsToIDs`'s keyset contain all the keywords used in the dump. It then iterates through the files that were not tagged with any keywords. For each of these files, if a description is provided, it checks to see if any of the words in the description was used explicitly as a keyword to tag other files (by consulting `TagsToIDs`'s keyset). If so, it creates an artificial set of keywords for the file comprising such words, and updates the three dictionaries accordingly. **Method 3** adds a file to the corpus if it has been tagged with keywords, using those keywords. If a file is not tagged with keywords, it then treats the words in the file's description, if it has any, as the file's keywords. Finally, **Method 4** treats both a file's keywords and the words in its description as keywords. The different methods are enumerated in order of increasing verbosity:

- **Method 1.** For each record R_i in the database, corresponding to the sound file F_i with file ID ID_i, check if F_i has been tagged with keywords (that is, the sixth field in R_i is not `\N`).

If F_i has been tagged with keywords $W = \{w_0, w_1, ..., w_n\}$, then remove stopwords from W, normalize the words in W, and set `IDsToTags`$[ID_i] = \{w_0, w_1, ..., w_n\}$. Add ID_i to `TagsToIDs`$[w_i]$ for $w_i \in W$, and set `IDsToMeta`$[ID_i] = R_i$.

If F_i has not been tagged with keywords, do not change any of the dictionaries.

- **Method 2.** Do **Method 1** first.

  Then for each record R_i in the database, corresponding to the sound file F_i with file ID ID_i, such that F_i is not tagged, check if a description is provided for F_i (that is, the sixth field in R_i is not an empty string).

  If a description is provided, extract the set of words $Desc = \{desc_0, desc_1, ..., desc_n\}$ from the description by removing special characters and splitting the description around empty spaces. Remove stopwords from $Desc$ and normalize the words in $Desc$. Create a reduced set of words $D = \{d_0, d_1, ..., d_m\} \subseteq Desc$ such that a word $desc_i$ from $Desc$ is in D iff `TagsToIDs`$[desc_i] \neq \emptyset$. Set `IDsToTags`$[ID_i] = \{d_0, d_1, ..., d_m\}$. Add ID_i to `TagsToIDs`$[d_i]$ for $d_i \in D$, and set `IDsToMeta`$[ID_i] = R_i$.

  If a description is not provided (thus the file has been neither tagged nor described), do nothing.

- **Method 3.** Do **Method 1** first.

  Then for each record R_i in the database, corresponding to the sound file F_i with file ID ID_i, such that F_i is not tagged, check if a description is provided for F_i.

  If a description is provided, extract the set of words $Desc = \{desc_0, desc_1, ..., desc_n\}$ from the description by removing special characters and splitting the description around empty spaces. Remove stopwords from $Desc$ and normalize the words in $Desc$. Set `IDsToTags`$[ID_i] = \{desc_0, desc_1, ..., desc_n\}$. Add ID_i to `TagsToIDs`$[desc_i]$ for $desc_i \in Desc$, and set `IDsToMeta`$[ID_i] = R_i$.

If a description is not provided (thus the file has been neither tagged nor described), do nothing.

- **Method 4.** For each record R_i in the database, corresponding to the sound file F_i with file ID ID_i, initialize a set K of keywords for F_i to $\{\}$.

  Check if F_i has been tagged with keywords. If F_i has been tagged with keywords $W = \{w_0, w_1, ..., w_n\}$, then remove stopwords from W, normalize the words in W, and add $w_i \in W$ to K.

  Check if a description is provided for F_i. If a description is provided, extract the set of words $Desc = \{desc_0, desc_1, ..., desc_n\}$ from the description by removing special characters and splitting the description around empty spaces. Remove stopwords from $Desc$, normalize the words in $Desc$, then add $desc_i \in Desc$ to K.

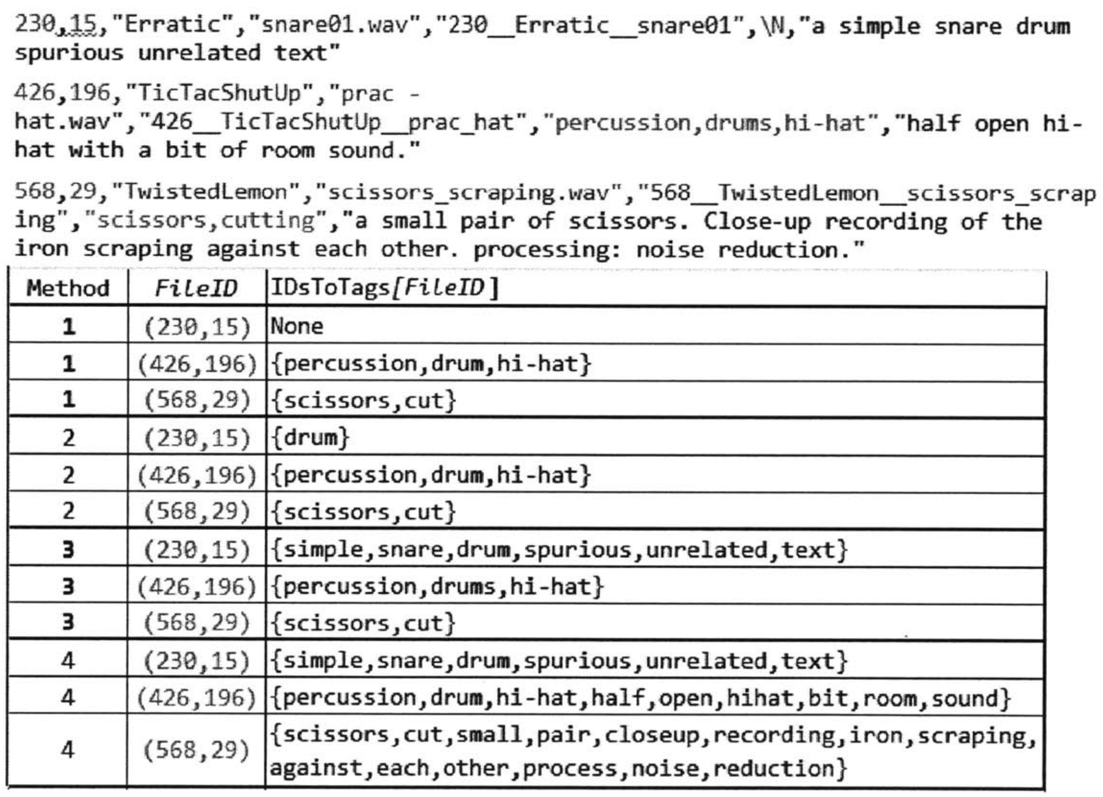
  If $K = \{k_0, k_1, ..., k_m\}$ is not empty, then set `IDsToTags`$[ID_i] = \{k_0, k_1, ..., k_m\}$. Add ID_i to `TagsToIDs`$[k_i]$ for $k_i \in K$, and set `IDsToMeta`$[ID_i] = R_i$.

  If K is empty, do nothing.

In all of the methods, files that are neither tagged nor described are treated as if they do not exist. This is because the only other field that can possibly indicate the content in the file is the original file name (the fourth field), but filenames are very often ill-indicative of content and hard to parse.

Figure 3-3 is an example of the `IDsToTags` dictionary entries after running each of the methods, if the database dump contained only the three records shown at the top of the figure, and highlights the strengths and weaknesses of each approach:

- **Method 1** skips any file that hasn't been tagged with keywords, but has been described. This is likely the correct behavior if the uploader, who did not tag the file but described it, thought that none of the words in the description were strongly associated enough with the file to be considered "keywords." However, it is more likely that the uploader was simply lazy and did not take the time to provide both keywords and a description. For example, keywords in the



```
230,15,"Erratic","snare01.wav","230__Erratic__snare01",\N,"a simple snare drum
spurious unrelated text"

426,196,"TicTacShutUp","prac -
hat.wav","426__TicTacShutUp__prac_hat","percussion,drums,hi-hat","half open hi-
hat with a bit of room sound."

568,29,"TwistedLemon","scissors_scraping.wav","568__TwistedLemon__scissors_scrap
ing","scissors,cutting","a small pair of scissors. Close-up recording of the
iron scraping against each other. processing: noise reduction."
```

Method	*FileID*	IDsToTags[*FileID*]
1	(230,15)	None
1	(426,196)	{percussion,drum,hi-hat}
1	(568,29)	{scissors,cut}
2	(230,15)	{drum}
2	(426,196)	{percussion,drum,hi-hat}
2	(568,29)	{scissors,cut}
3	(230,15)	{simple,snare,drum,spurious,unrelated,text}
3	(426,196)	{percussion,drums,hi-hat}
3	(568,29)	{scissors,cut}
4	(230,15)	{simple,snare,drum,spurious,unrelated,text}
4	(426,196)	{percussion,drum,hi-hat,half,open,hihat,bit,room,sound}
4	(568,29)	{scissors,cut,small,pair,closeup,recording,iron,scraping, against,each,other,process,noise,reduction}

Figure 3-3: Example IDsToTags Dictionary

record for File (230,15) in Figure 3-3 are completely skipped, even though its description, "a simple snare drum spurious unrelated text," contains useful information. The only possible benefit to Method 1 is that the resulting corpus is smaller and thus faster to use.

- **Method 2** skips any file that has been neither tagged nor described. It also skips any file that has been described and not tagged, if none of the words in the description are pre-existing tags. As shown in Figure 3-3, after using Method 2, File (230,15) is included with the single tag "drum." The extra descriptor "snare," although useful, is not included as a tag because it is not explicitly used as a tag for Files (426,196) or (568,29).

- **Method 3** does not skip any file with metadata, but uses only either the tags or the description. However, if it uses the description, it uses every (non-stopword) word in the description. Many records in the database contain overly long descriptions with spurious unrelated text, such as File (230,15) in Figure 3-3. Consequently, these spurious, unrelated words show up as tags in the corpus (for example, the tags for File (230,15) using this method are "{simple,snare,drum,spurious,unrelated,text}").

- **Method 4** does not skip any file with metadata, and uses both the tags and description. Thus, if a file was uploaded by a user who put only the most relevant words in the tags, and provided a much more eloquent and accurate description, this method might be the correct behavior. However, it has the same weakness to spurious, unrelated descriptive text that **Method 3** does, so the tags in the corpus for Files (230,15) and (586,29) both include spurious, unrelated words.

3.5.2 Sound Corpora Comparison

The next step after generating the corpora using the methods described in the previous section was to choose which among the four to use.

Method	# Unique Tags	# Files Added to Corpus	Average Tags/File	Average Files/Tag
1	18044	72318	5.81	23.27
2	18044	72882	5.77	23.50
3	18414	72884	5.77	23.05
4	40713	72900	5.78	25.95

Figure 3-4: Corpus Analytics

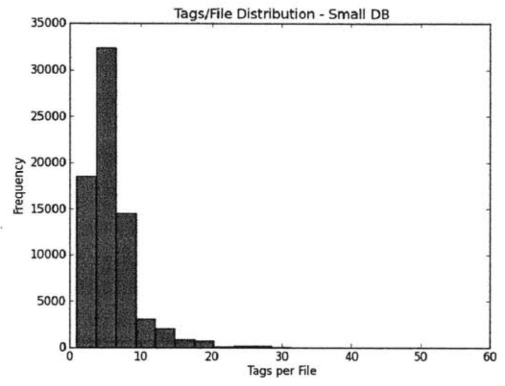
Figure 3-4 shows high level statistics of the `TagsToIDs` and `IDsToTags` dictionaries using each of the four methods in the previous section with the Freesound database dump. The original dump contains 72,999 files, 99 of which were uploaded with neither tags nor description.

Method 2 includes 564 files from the original dump that Method 1 omits. These files are untagged but contain known tag words in their descriptions. Method 3 includes only 2 new files from the original dump that Method 2 does not include, but introduces 370 new tags. Finally, Method 3 includes only 16 new files from the original dump that Method 2 does not include, but introduces 22,299 new tags.

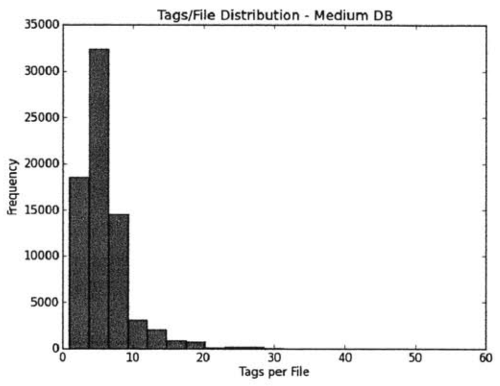
As mentioned earlier, the only benefit of Method 1 over Method 2 was the potential for faster computation. However, only 564 extra files out of 72,999 are added in Method 2, so any detraction from performance would be hardly noticeable. Consequently, the task became choosing the best of Methods 2, 3, and 4.

Figures 3-5a, 3-5b, and 3-5c present the distribution of files by number of tags per file. No file is tagged with less than one keyword (by construction), no file is tagged with more than 60 keywords (by nature of the original Freesound database), and as mentioned earlier, each file is tagged with an average of 5-6 keywords. Because there is a total increase of 386 originally non-included files from Method 2 through Method 4, the change in distributions across the methods is very slight.

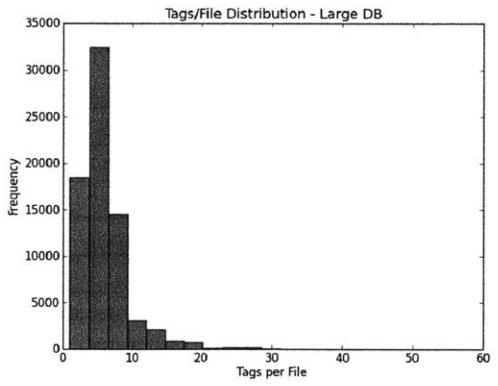
Figures 3-6a, 3-6b, and 3-6c present the distribution of keywords by number of files tagged per keyword. No keyword is used to tag less than one file (by construction), no keyword is used to tag more than 18,381 files (by nature of the original Freesound database), and as mentioned earlier, a keyword is used to tag, on average, between



(a) Method 2

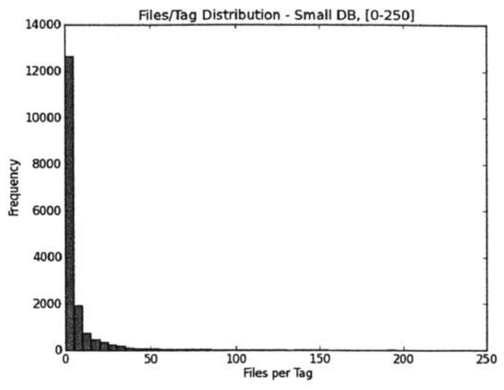


(b) Method 3

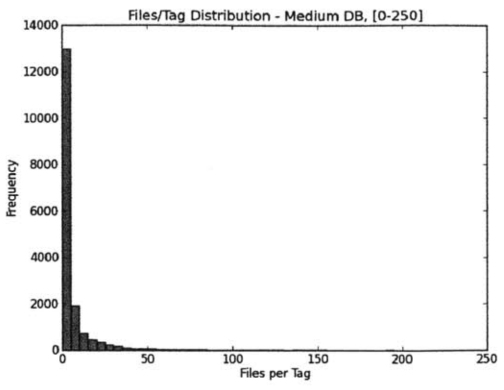


(c) Method 4

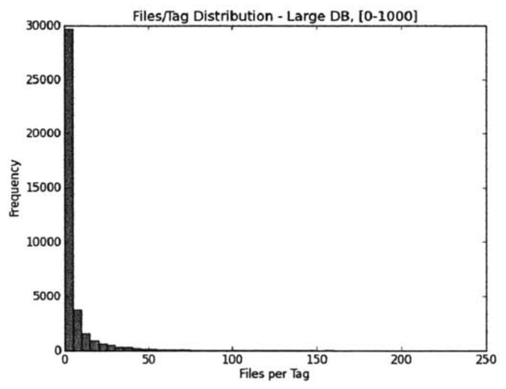
Figure 3-5: Tags/File Distribution for Different Sound Corpora



(a) Method 2



(b) Method 3



(c) Method 4

Figure 3-6: Files/Keyword Distribution for Different Sound Corpora

23 and 26 distinct files. Although the maximum frequency of a keyword is 18,381, figures 3-6a, 3-6b, and 3-6c only show the portions of the distributions for keywords that occur with frequencies between 0 and 250. Since 95% of keywords are occur with frequencies not exceeding 74, by any of the methods, the truncated distributions contain the vast majority of the whole distributions while allowing a more detailed look at the part of the distributions which experience the most variation across methods.

Of the new keywords introduced to the corpora by Methods 3 and 4, most are used to tag no more than five unique files, implying that the new keywords are infrequently used among the descriptions for all the files in the corpus. Some further exploration suggests that the newly added keywords are mainly spurious and unrelated to the actual sound content, such as the brand name of the sound recording device used, the name of the upload, or abbreviations. Figure 3-2 provides a few examples of spurious and unrelated descriptors that are included as tags by Methods 3 and 4, such as "David," "Bithell," "Ali," "sony," "esm-ds-70p," and "gt." From an intuitive standpoint, it makes sense that most of the new keywords are unrelated to the actual sound content – 72,318 of the 72,999 sound files are used to generate the original set of keywords in Method 2, and the new keywords in Methods 3 and 4 are drawn from the descriptions of no more than 16 files or from the descriptions of files that have been tagged with keywords. It would be highly unlikely that relevant and useful keywords derived in this manner have not already been used as keywords previously.

Because Methods 3 and 4 introduce a very large number of keywords that are unrelated or not useful to the corpus generated by Method 2, and do not introduce a large number of files that were excluded, we use the corpus generated by Method 2 in the Narratarium system.

3.6 SoundGuesser Metrics Implementation

It was necessary to define a metric $C(H_s)$ that would evaluate `SoundGuesser`'s certainty in a particular sound file hypothesis H_s. We found that defining a metric which would evaluate how strongly a word was associated with the concept of sound in the first place, if used in conjunction with the guessing algorithms, would lead directly to $C(H_s)$. We call this metric "soundiness," and drew inspiration for its implementation from Colorizer's voting algorithm.

3.6.1 Colorfulness

Given a word W, Colorizer generates a color hypothesis for W by first creating a set of color hypotheses. This set contains not only the colors that W itself maps to in the color corpus, but also the colors that the 5 most related words to W, according to Divisi's relatedness matrix *divisi*, map to. Each color C in the set is weighted as follows:

$$w(C) = I(C, W) + \sum_{K \in R} I(C, K) divisi[W, K] \tag{3.1}$$

where $I(C, M) = 1$ if C is in the set of colors that M maps to in the color corpus and 0 otherwise, and R is the set of the 5 most related words to W in *divisi* (*i.e.* R =*divisi.row_named(W).top_items(5)*. Then Colorizer runs a single transferable vote election using a small sample from the set of color hypotheses as candidates; each color hypothesis in the pool gives a weighted vote, with weight equal to its own weight, to the candidate in the set that is the smallest Euclidean distance from it in the LAB color coordinate plane. The candidate that receives the most net votes is returned as the best hypothesis.

Although Colorizer returns only a RGB color hypothesis for a word W, its voting algorithm generates an implicit metric of certainty for the hypothesis – the amount of votes the winning candidate received, which is the sum of the $w(C)$ in Formula 3.1 for some subset of the colors that W and *divisi.row_named(W).top_items(5)* map to in the color corpus. This metric can be interpreted as the "colorfulness" of W in that it not only indicates the strength of the color hypothesis, but also the extent to which W and the words most related to it are associated with the concept of color in the first place. If W and the words related to it map to a very large set S of colors in the color corpus, then the value of the metric will be high, regardless of the value of the RGB hypothesis, because it receives the majority vote from a very large set of voters. Since the color corpus was created using the color survey in [4], a word W will map to more colors if more people associated a color with it, regardless of what color any single person chose. Colorizer's logic thus relies on the (reasonable) assumption that if humans more commonly associate a word W_1 with the concept of color than another word W_2, then more colors will be associated with W_1 than with W_2.

3.6.2 Soundiness

In a similar fashion to Colorizer's voting method, the "soundiness" metric makes the assumption that if a word W_1 is more strongly associated with the concept of sound than another word W_2, it will be used to describe sound files more frequently. Then W_1 will appear as a keyword more frequently in Freesound's public sound database, and thus `SoundGuesser`'s corpus, than W_2. Simply defining the soundiness of a word W as its frequency as a keyword in the sound corpus, however, forces words that are strongly associated with sound, but not commonly used in speech, such as "bassoon," "piccolo," and "timpani," to have misleadingly low scores. We thus use the three most semantically related words to W (*i.e.* *divisi.row_named(W).top_items(3)*) to generate the metric (similar to how Colorizer uses the five most semantically related word, in addition to the word itself). We tried three different methods of measuring the soundiness *Soundiness* of a word W.

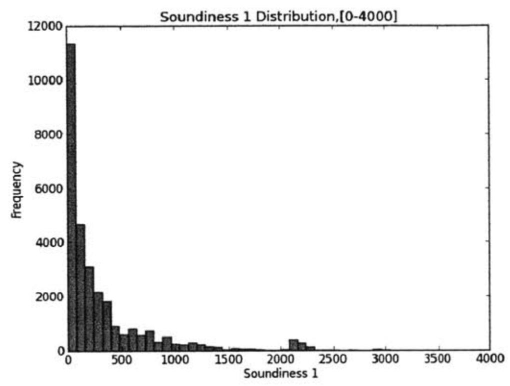
- **Method 1.** Initialize a set $Matches$ of sound file IDs to $\{\}$. Then for each word W_i in *divisi.row_named(W).top_items(3)*, set $Matches = \texttt{TagsToIds}[W_i] \cup Matches$. Return $Soundiness(W) = |Matches|$.

- **Method 2.** Initialize $Soundiness(W) = 0$. Then for each word W_i in *divisi.row_named(W).top_items(3)*, set $Soundiness(W) = Soundiness(W) + |\texttt{TagsToIds}[W_i]|$. Return $Soundiness(W)$.

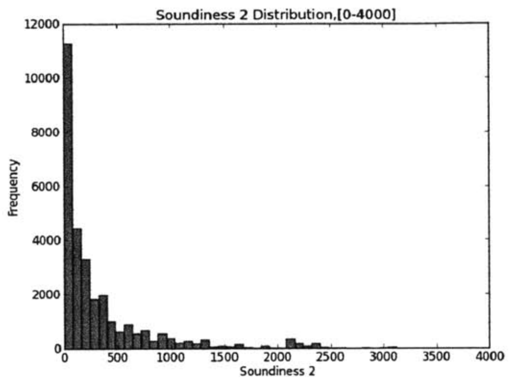
- **Method 3.** Initialize $Soundiness(W) = 0$. Then for each word W_i in *divisi.row_named(W).top_items(3)*, set $Soundiness(W) = Soundiness(W) + divisi[W, W_i] \cdot |\texttt{TagsToIds}[W_i]|$. Return $Soundiness(W)$.

Figures 3-7a, 3-7b, and 3-7c show the distribution of keywords for soundiness values using the three methods described above. Although some keywords have soundiness values in excess of 4,000, these comprise less than 5% of the total keywords in the corpus(regardless of method), so the distributions are truncated to better illustrate the sections of greatest change across distributions.

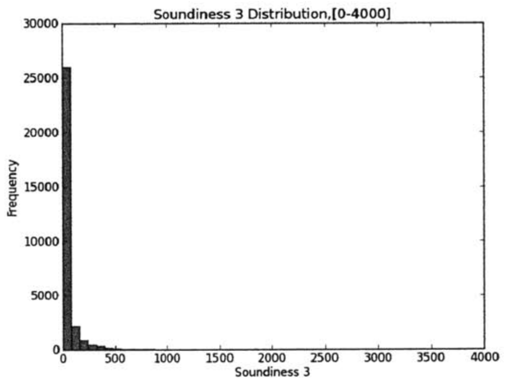
Methods 1 and 2, corresponding to the distributions in Figures 3-7a and 3-7b, generate distributions with higher variance than the distribution generated by Method 3. This is a desirable characteristic because differences between sound-related words and sound-unrelated words are exaggerated. Because the differences in the



(a) Method 1



(b) Method 2



(c) Method 3

Figure 3-7: Keyword Distribution for Different Soundiness Metrics

distributions generated by Methods 1 and 2 were slight, we use Method 2 in our implementation because there is slightly more variation in its distribution.

3.7 SoundGuesser Guessing Algorithm Implementation

The sound hypothesis for a word W_n needs to be context aware, accompanied with a measure of certainty, and generated in real-time with low latency. These three requirements are not independent of each other. A hypothesis is more certain and accurate when the algorithm used to generate it is more context aware, both in terms of considering words previously entered by the user, and in terms of considering words other than the input that are semantically related to the input. However, increasing the certainty and accuracy of a hypothesis by increasing the algorithm's context awareness will also increase the amount of necessary computation, thus adversely affecting latency.

Because the computer that will be used to run Narratarium is unknown, there are no guarantees regarding processing speed. We have included a high degree of customization in our algorithms to deal with this uncertainty, and present performance results in the following chapter.

We implemented a number of different guessing algorithms, which can be conceptually reduced to a naive algorithm, a graph-based algorithm, and several variations on the latter. We describe them in the following sections.

Each of the guessing algorithms is independent of the others. When `Reader` is initialized upon Narratarium's launch, it creates an instance of the `SoundGuesser` subclass (see Figure 3-1), with the particular guessing algorithm determined by a configurable value. The instance of the `SoundGuesser` subclass then loads the sound corpus (the dictionaries `TagsToIDs`, `IDsToTags`, and `IDsToAllMeta`) and Divisi matrix from locally saved files, and initializes a few variables to maintain state:

- `maxHistory`, a configurable integer. The guessing algorithm will consider the last `maxHistory` words entered by the user when deriving a sound file hypothesis.

- `wordBuffer`, an array that keeps the last `maxHistory` words passed to `SoundGuesser`, with the most recent word at the front. Initially empty.
- `branch`, a configurable integer. The guessing algorithm will consider, in addition to each word W in `WordBuffer`, the `branch` most semantically related words to W (*i.e. divisi.row_named(W).top_items(branch)*)
- `maxDepth`, a configurable integer. This will be used by the graph algorithms described later.
- `decayRate`, a configurable float between 0.0 and 1.0. Given `wordBuffer`$=$ $[w_0, w_1, ..., w_n]$, the guessing algorithm will consider w_k to be `decayRate` as important as w_{k+1} for $k \in [0, 1, ..., n-1]$ when deriving the hypothesis.
- `hypothesis`, a tuple which includes the file ID of the most recent hypothesis and the hypothesis's certainty. This is initially null.
- `NetworkMap`, a dictionary which represents a graph network. This will be used by the graph algorithms described later.

When `SoundGuesser` is passed a word by `Reader`, it updates `wordBuffer` accordingly by appending the word to the front of `wordBuffer` and popping the last value from `wordBuffer` if `wordBuffer` exceeds `maxHistory` in length. It then calls the guessing algorithm, which updates `hypothesis`.

3.7.1 Naive Algorithm

The algorithm first creates an empty dictionary mapping file IDs to scores, *soundMatches*, which will be used to keep track of any sound file that could potentially be associated with the words in `wordBuffer`, as well as the algorithm's confidence in its appropriateness. *soundMatches* is implemented as a Python default dictionary, so that $soundMatches[x] = 0$ if $x \notin soundMatches$.

Then, the algorithm iterates through the words in `wordBuffer` starting with the most recently entered word at `wordBuffer`[0]. For each word W_i with index i in `wordBuffer`, it performs the following steps:

1. Create a default dictionary D and set $D[W_i] = 1.0$.

2. Create $S = [(w_0, p_0), (w_1, p_1), ..., (w_{branch-1}, p_{branch-1})]$ where w_j is the $(j+1)$th most semantically related word to W_i using *divisi*, and p_j is the relatedness coefficient $divisi(W_i, w_j)$.

3. Normalize the p values in S by dividing each p_j by $\sum_{k=0}^{branch-1} p_k$ for $j \in [0, 1, ..., branch-1]$. Then, multiply each p_j in S by $\texttt{decayRate}^i$ for $j \in [0, 1, ..., branch-1]$.

4. Set $D[w_j] = D[w_j] + p_j$ for all $(w_j, p_j) \in S$.

5. For each $(w_j, p_j) \in D$:

    1. Create a set of file IDs that w_j maps to in the sound corpus, $F[0...m] = \texttt{TagsToIds}[w_j]$.

    2. Set $soundMatches[F[n]] = soundMatches[F[n]] + Soundiness(W_i) \cdot p_j$ for $n \in [0, 1, ...m]$.

At this point, the algorithm has iterated through each word in `wordBuffer` and, for each word, found a set of sound files that could possibly be associated with the word. For each of these sound files, it adds the sound file to *soundMatches* if it is not already in *soundMatches*, then increases its weight by a value that can be interpreted as the word's vote (thus each word can vote for any number of files). The vote's weight is a function of the soundiness of the word, the recency of the word, the decay rate, and the relatedness of the word to the sound file.

Finally, the algorithm sorts *soundMatches* by weight, and extracts the file ID with maximum weight. It sets `hypothesis` to a tuple comprising these two values, and returns it. The algorithm's certainty in the hypothesis is equivalent to the hypothesis's weight.

3.7.2 Graph Algorithm

Theory

The theory behind this algorithm is similar to that of "activation spreading" mentioned in [8]. If a graph algorithm is used, `SoundGuesser` uses Divisi's relatedness graph (the undirected graph whose edge weights are contained in the relatedness matrix *divisi*) and the sound corpus to create a new graph stored in `NetworkMap`. It first replicates *divisi*. Then for each file ID *id* in the sound corpus, it creates a new node named *id* and constructs an undirected edge of weight 1.0 between each keyword in `IDsToTags`[*id*] and the node *id*. Thus `NetworkMap` comprises "keyword" nodes, or nodes corresponding to the words in *divisi*, "file" nodes, or nodes corresponding to the files in the sound corpus, and undirected edges. The algorithm then augments file nodes with a value called "energy," initially set to zero. The energy of a file node represents the system's current estimate of the file's appropriateness for the words in `WordBuffer` (or, the algorithms' certainty in the particular file), and the file ID of the file node with the highest energy is considered the best hypothesis.

When a new word is added to `WordBuffer`, the graph algorithm is called to update `NetworkMap`. The algorithm first decays the energy in the system by `decayRate`. Then, it introduces energy equal to the soundiness of the word to the system. The energy flows along edges through keyword nodes and is deposited into file nodes. When energy is transmitted along an edge, its magnitude is extenuated by the weight of the edge. The algorithm can also handle the removal of a word W from `WordBuffer` by spreading negative weight through the system to counteract the effects of W's earlier addition.

Implementation

Because *divisi*'s underlying graph has edges between each of its 66,375 nodes, and the sound corpus contains 72,882 files, each of which is tagged 5.77 times on average, it is impossible to implement the algorithm in full while maintaining low latency because the graph is simply too large. The `SoundGuesser` variables `branch` and `maxDepth` are thus used to approximate the algorithm by limiting the number of times energy is spread through the system.

The differences between the implemented algorithm and the theoretical algorithm are as follows:

- `NetworkMap` is created lazily rather than upon initialization, and only contains file nodes. It is initialized as an empty default dictionary.

- A helper method called $spreadEnergy(keyword, energy, branch, depth)$ spreads energy from the keyword node *keyword* with value *energy.* It follows two steps:

  1. Transmits energy to the file nodes to which *keyword* is adjacent by setting `NetworkMap`$[id]$ = `NetworkMap`$[id] + energy$ for each *id* in `TagsToIds`$[keyword]$.

  2. If $depth > 0$, spreads energy to the keyword nodes to which *keyword* is adjacent. The amount of spreading is limited here by only spreading energy along the `branch` edges with the highest weights. Thus the method creates a set $S = [(w_0, p_0), (w_1, p_1), ..., (w_{branch-1}, p_{branch-1})]$ where w_j is the $(j+1)$th most semantically related word to *keyword* using *divisi*, and p_j is the relatedness coefficient $divisi(keyword, w_j)$, and normalizes the p values in the same manner as the naive algorithm. Then for each w_j in S, the method recursively calls $spreadEnergy(w_j, energy \cdot p_j, branch, depth - 1)$.

- When a new word W is added to `WordBuffer` and the algorithm is called to update `hypothesis`, it calls $spreadEnergy(W, Soundiness(W), branch, maxDepth)$. It then sorts `NetworkMap` by energy, extracts the file ID with the highest energy, and updates `hypothesis` accordingly. The algorithm's certainty in the hypothesis is equivalent to the file's energy.

3.7.3 Graph Algorithm Variations

The graph algorithm described previously exhibits two important characteristics:

- It updates NetworkMap incrementally. When WordBuffer is not full and a new word W is added, SoundGuesser will call the algorithm once to perform $spreadEnergy(W, Soundiness(W), branch, maxDepth)$. When WordBuffer is full and a new word W is added, an old word W' is necessarily removed. Thus SoundGuesser will call the algorithm twice - first to perform $spreadEnergy$ $(W, Soundiness(W), branch, maxDepth)$ and second to perform $spreadEnergy$ $(W', -Soundiness(W) \cdot decayRate^{maxHistory-1}, branch, maxDepth)$. Note that during the algorithm's second call, the energy in NetworkMap is not decayed.

- It allows energy that was spread from a keyword node W_0 to be spread back to W_0. For example, if branch=3 and the word "band" is added, energy will be spread from the node for "band" to the nodes for "instrument," "orchestra," and "saxophone." However, the energy which is spread to "instrument" will be recursively spread back to the nodes for "band" and "saxophone" (in addition to the node for "violin").

We implemented three variations of the algorithm to explore the effect of these characteristics. The first variation updates NetworkMap incrementally, but does not allow energy to be transmitted back into keyword nodes that have already received energy during the same call to the algorithm (that is, if branch=3 and the word "band" is added, energy will be spread from "band" to "instrument," "orchestra," and "saxophone," then from "instrument" to "violin," "guitar," and "musician"; if the word "band" is added again, energy will be spread to the same nodes). The second variation resets NetworkMap and performs an add operation for each word in WordBuffer, allowing backwards transmission of energy. The third variation resets NetworkMap and performs an add operation for each word in WordBuffer, not allowing backwards transmission of energy.

If the graph algorithm is called to update hypothesis every time a new word is added to WordBuffer, the original algorithm and the first variation will, on average, be much more efficient than the second and third variations. However, considering that the sound files in FreeSound's database can last tens of seconds, and that users can enter or speak several words in that amount of time, a new hypothesis may not be necessary for every word entered. Because the original algorithm and the first variation are incremental, they must be called every time a word is entered, and so there is an average of two **maxDepth** energy spreads per word. Thus, if a hypothesis update is required less frequently than every $\frac{maxHistory}{2}$ words entered, the second and third variations will perform better on average.

This point is only relevant to the story mode experience, in which a story with pre-generated sound and color values is presented to the user. In freestyle mode, worst case performance is the only meaningful metric.

Chapter 4

Evaluation

4.1 Success Metrics

As mentioned in the previous chapter, the three metrics used to evaluate the back end are that the hypotheses are generated in a context-aware manner, the hypotheses are accompanied by a measure of certainty, and that the hypotheses are generated in real-time with low latency.

Because the back end guessing algorithms are designed with configurable levels of context-awareness and with the measure of certainty as the back end's means of distinguishing between hypotheses, the task of evaluation simplifies to evaluating the latency of the back end as a function of context-awareness.

4.2 Latency Evaluation

There are three ways to configure the level of context-awareness:

- Increase the value of `maxHistory`. This increases the maximum size of `wordBuffer` and forces the back end to consider more of the context of the story being told. From a practical point of view, one might expect the accuracy of hypotheses to experience dramatically diminishing marginal returns with `maxHistory` for two reasons - first, words that are separated from each other in a story by several intermediate words are much less likely to describe the same concept, sound, or color due to the inherent nature of storytelling. Second, the configurable `decayRate` field forces the back end to pay exponentially less attention to the less recently entered words. Assuming `decayRate`=0.75, increasing `maxHistory` past a value of 10 will have negligible effect on the hypothesis that is made, while having a detrimental effect on latency.

- Increase the value of `branch`. This increases the amount upon which the back end employs *divisi*'s commonsense knowledge when interpreting user input.

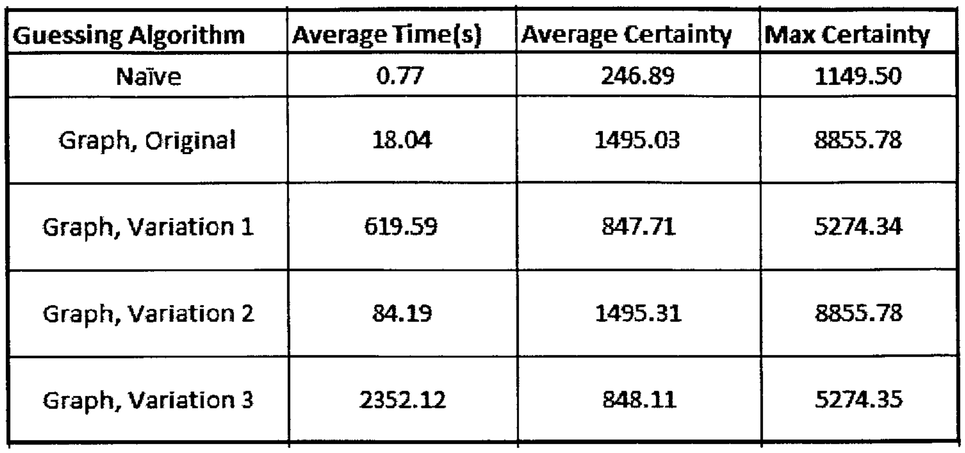
- Increase the value of `maxDepth`. This only applies to the graph-based algorithms, and similar to `branch`, increases the amount upon which the back end employs *divisi*'s commonsense knowledge when interpreting user input.

4.2.1 Experiment Design

We evaluated the latency of each of the guessing algorithms by varying `maxHistory`, `branch`, and `maxDepth`. We randomly generated a sequence of 15 words, drawn from the rows of Divisi's relatedness matrix. The experiment added the first word in the sequence, requested an updated hypothesis, measured the time the algorithm took to compute the hypothesis, then added the second word in the sequence and repeated the process until all 15 words had been added. We ran the experiment for values of `maxHistory` between 1 and 10, values of `branch` between 1 and 5, and values of `maxDepth` between 1 and 5. `decayRate` was set to 0.75.

4.2.2 Experiment Results

Figure 4-1 presents the high-level performance results of the experiment, marginalized over the choice of guessing algorithm (and marginalizing out the configurable fields). Unsurprisingly, the naive algorithm is by and large the fastest (and by and large accompanied with the smallest confidence values), and the incremental graph algorithms are substantially faster than their non-incremental counterparts. Furthermore, the graph algorithms which allowed backwards spreading of energy performed between



Guessing Algorithm	Average Time(s)	Average Certainty	Max Certainty
Naïve	0.77	246.89	1149.50
Graph, Original	18.04	1495.03	8855.78
Graph, Variation 1	619.59	847.71	5274.34
Graph, Variation 2	84.19	1495.31	8855.78
Graph, Variation 3	2352.12	848.11	5274.35

Figure 4-1: High-Level Performance Results of Different Guessing Algorithms

3.8 and 4.7 times faster, on average, than the graph algorithms which did not allow backwards spreading of energy, while offering certainty measures approximately 1.76 times higher, on average. It is to be expected that allowing for backwards spreading of energy will reduce latency, since less keyword and file nodes are being explored and less file nodes are being added to `NetworkMap`. The increase in certainty seems to verify that the more common English words are also more common keywords, but the magnitude of the increase is highly dependent on the input word sequence and cannot safely be used to draw conclusions.

Figure 4-2 presents the average computation time for the incremental graph guessing algorithm which allows backward spreading of energy (the original graph algorithm), marginalized over choice of `maxHistory`. This result may seem somewhat surprising at first, but is actually rather intuitive. Given that fifteen words were used as input, a `maxHistory` value of 1 would require words to be removed from the buffer, and thus a second call to spread negative energy, after only one word was entered, and so the total number of algorithm update calls would be $1 + 2(14)$. Increasing `maxHistory` by 1 would put off the necessity for a second call to spread negative energy by one input word, so the total number of algorithm update calls would be $1+2(13)$. This effect will decrease as the number of input words increases, and since stories are often hundreds or thousands of words long, we expect that the choice of `maxHistory` will have no effect on performance due to the benefit of incremental

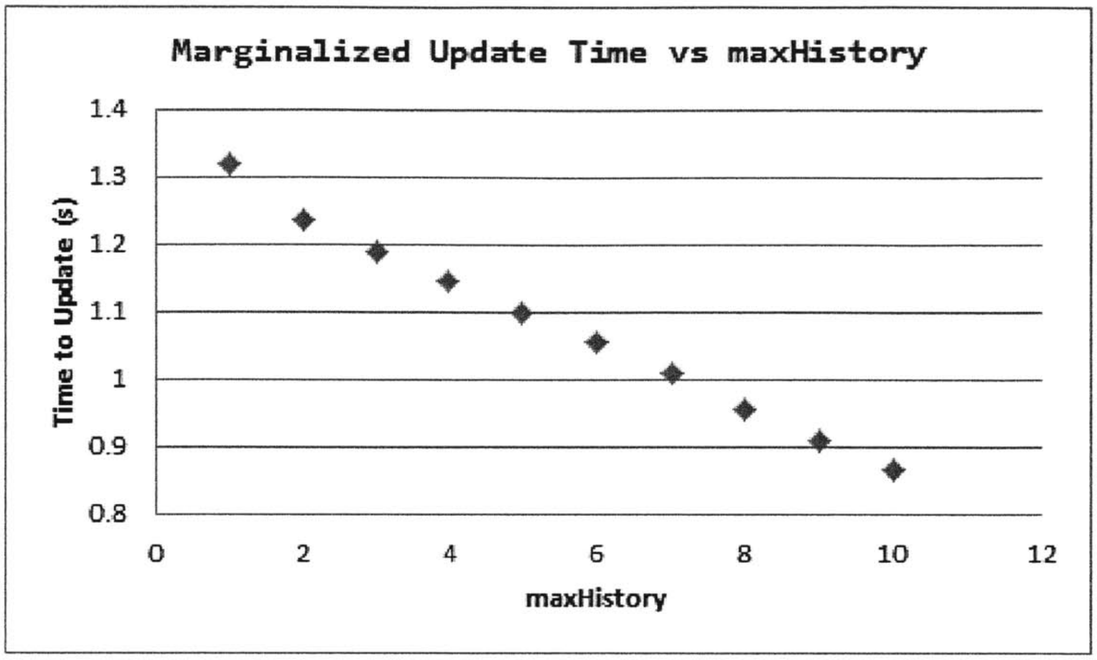


Figure 4-2: Average Performance of Graph Guessing Method, Marginalized over maxHistory

updating.

Figure 4-3 presents the average computation time for the incremental graph guessing algorithm which allows backward spreading of energy, marginalized over choice of `branch`. As should be expected, computation time increases exponentially with the branch factor.

Figure 4-4 presents the average computation time for the incremental graph guessing algorithm which allows backward spreading of energy, marginalized over choice of `maxDepth`. As should be expected, computation time increases exponentially with maximum depth of energy spread.

We found that, using the original graph guessing algorithm (which was used to generate Figures 4-2, 4-3, and 4-4), latencies only began to exceed 0.50 seconds when either of the values `branch` or `maxDepth` was set to 3 and the other of the values was set to a number greater than 3.

Figure 4-5 presents the average computation time for the complete graph guessing algorithm which allows backward spreading of energy, marginalized over `maxHistory`.

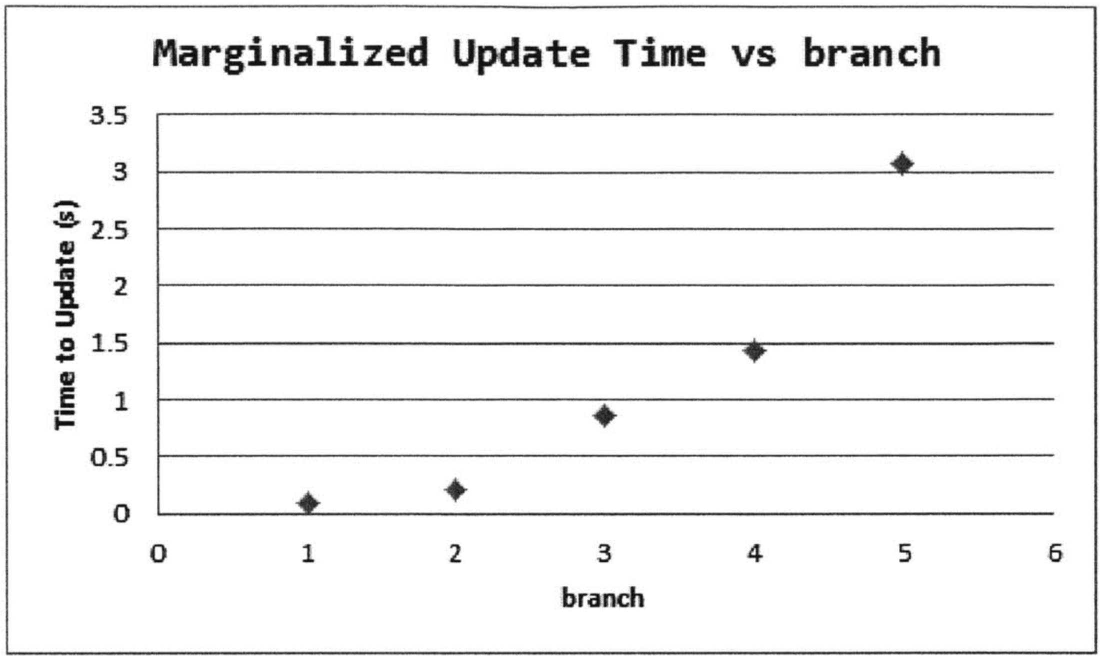


Figure 4-3: Average Performance of Graph Guessing Method, Marginalized over branch

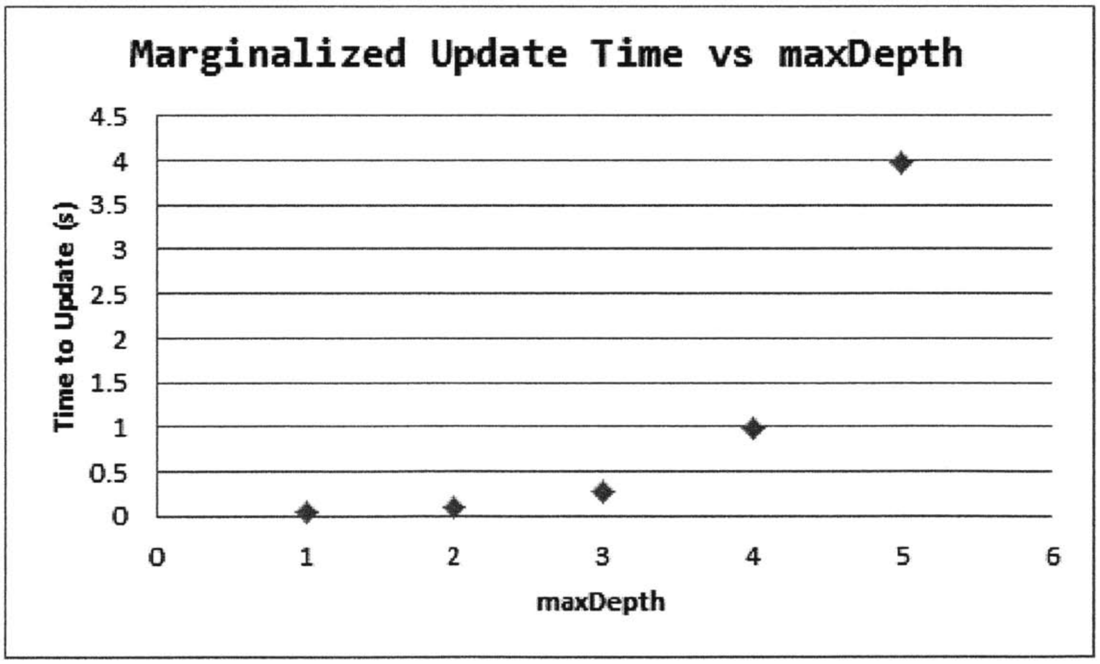


Figure 4-4: Average Performance of Graph Guessing Method, Marginalized over maxDepth

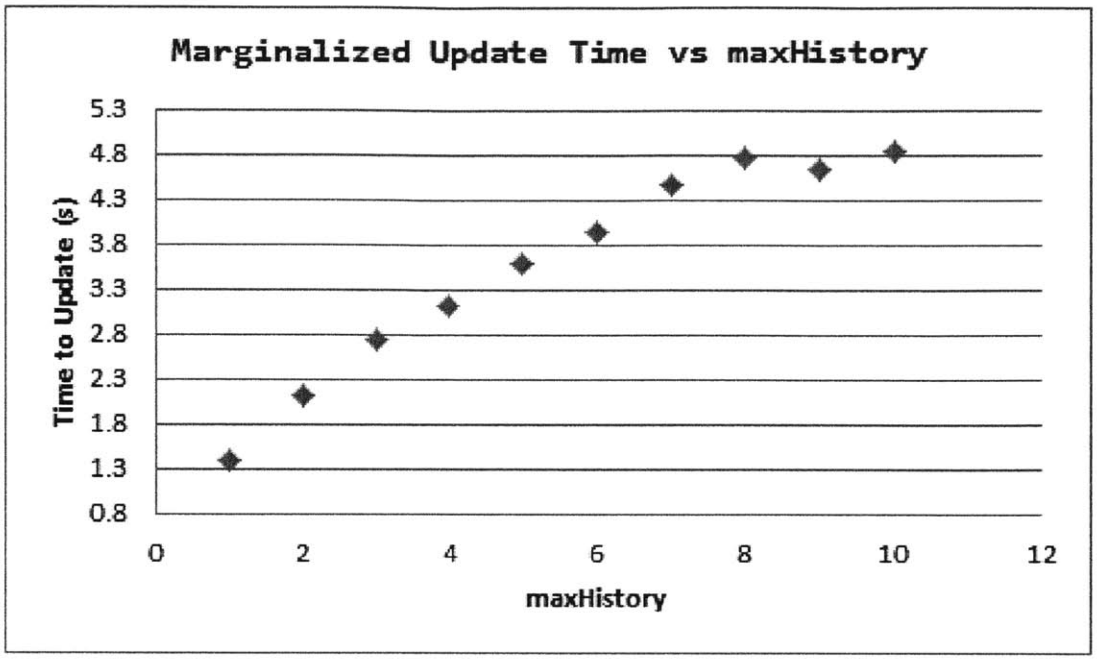


Figure 4-5: Average Performance of Graph Guessing Method Variant 2, Marginalized over maxHistory

As expected, the average marginalized update time varies almost linearly with `maxHistory` because this variation of the algorithm resets `NetworkMap`(the energy map) upon every new input word, and thus must perform n energy spreads, where n is the number of words in `WordBuffer`.

Figures 4-6 and 4-7 present the average computation time for the complete graph guessing algorithm which allows backward spreading of energy, marginalized over `branch` and `maxDepth`, respectively. As expected, computation time increases exponentially with both `branch` and `maxDepth`.

The performance graphs for the other two graph variants (incremental updates without backward spreading and complete updates without backward spreading) can be found in the appendix. They exhibit similar trends to their backward spreading counterparts here, and the only significant difference is a constant factor slowdown in computation time, attributable to the inherent need to search more keywords (and thus also more files).

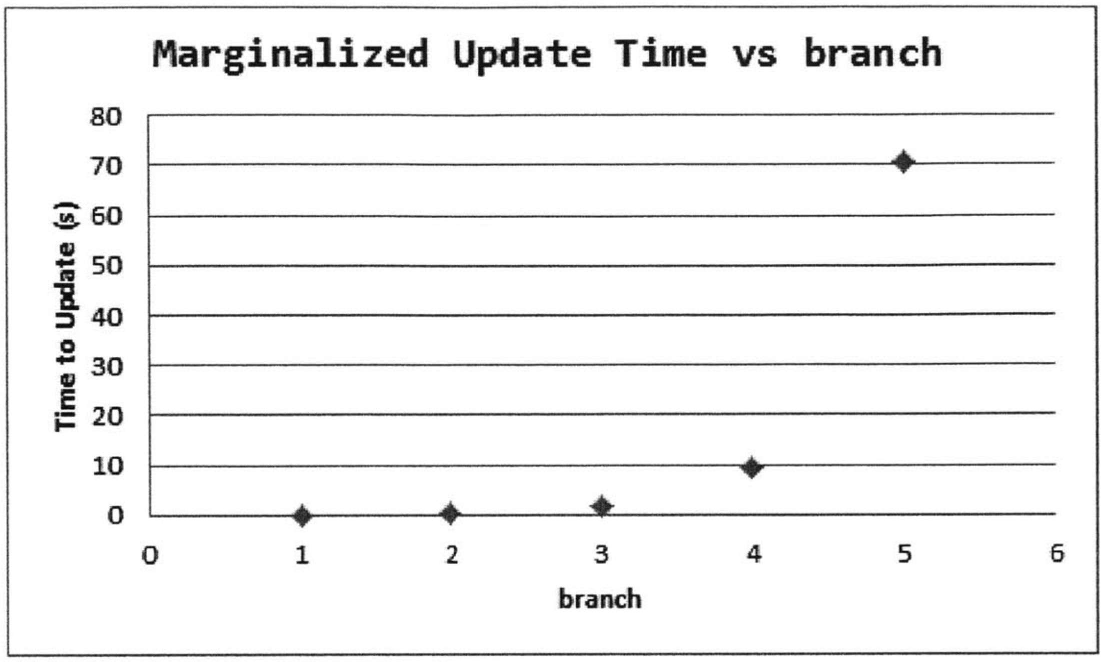


Figure 4-6: Average Performance of Graph Guessing Method Variant 2, Marginalized over branch

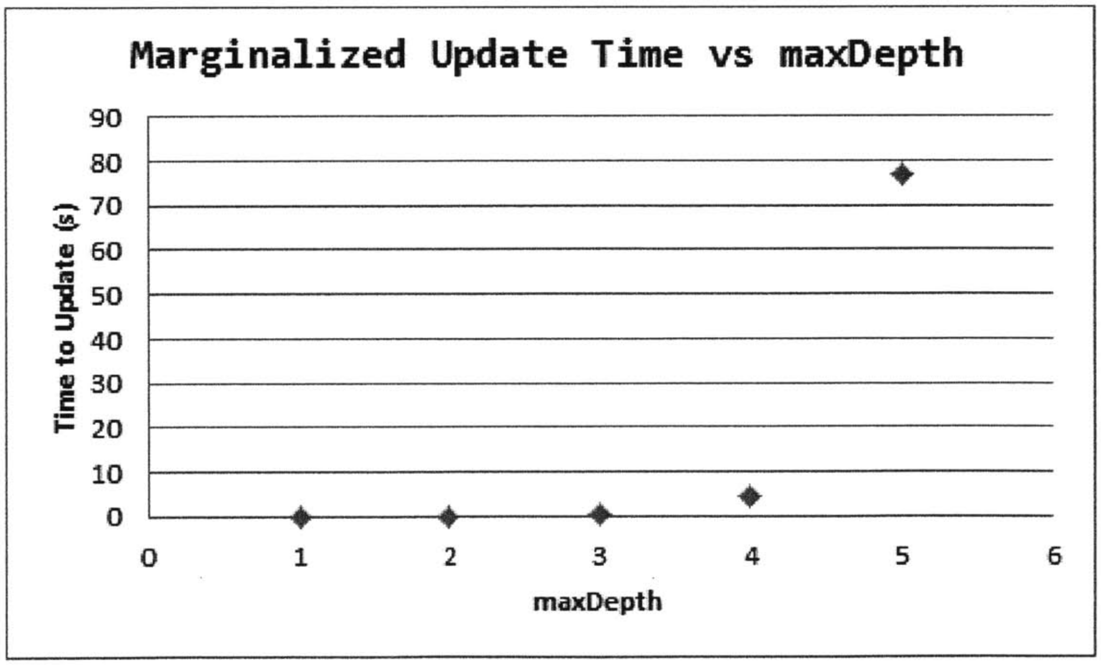


Figure 4-7: Average Performance of Graph Guessing Method Variant 2, Marginalized over maxDepth

4.3 Correctness

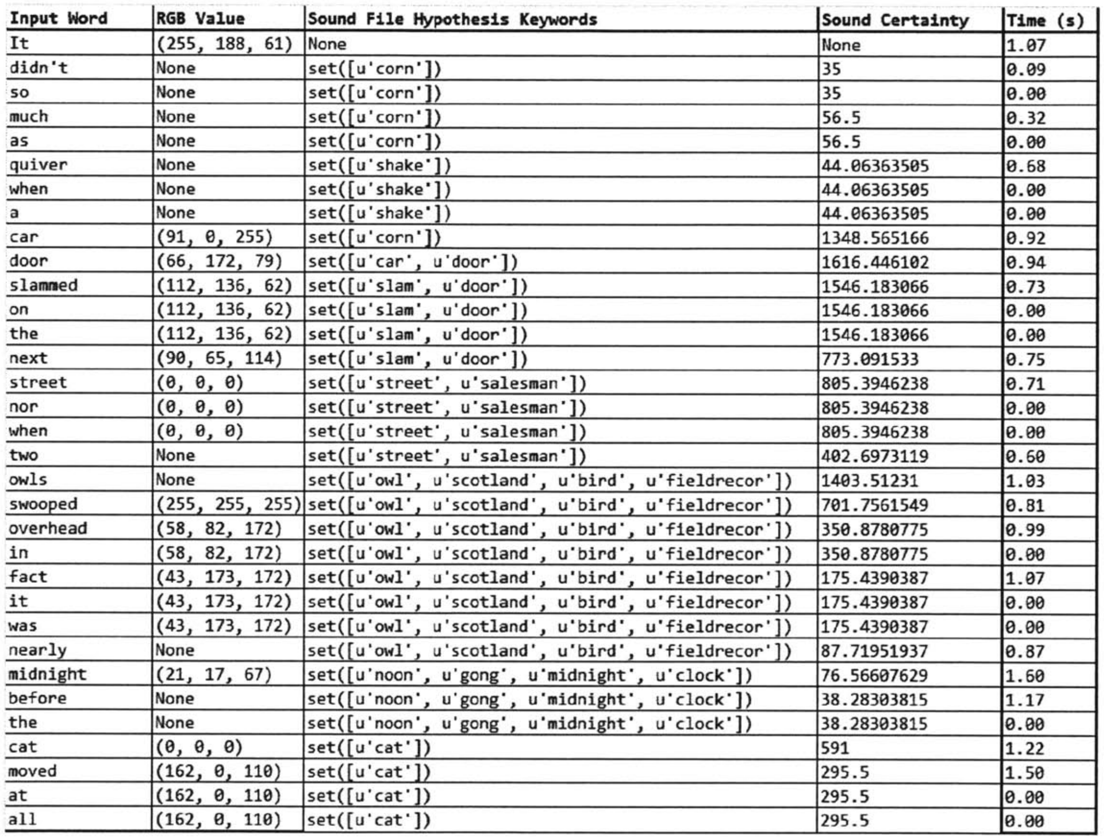
As mentioned in previous chapters, there is no objective or correct way to measure the correctness or appropriateness of hypotheses generated by Narratarium, because correctness is defined by the user's personal experiences and interpretation of the story. Furthermore, the hypotheses that Narratarium generates are not only dependent on its algorithms, but also on the corpora from which it draws sound and color guesses, the semantic relatedness matrix *divisi*, and Narratarium's many configurable parameters.

Figure 4-8 presents the results of running the back end update algorithm on the two sentences,

> *It didn't so much as quiver when a car door slammed on the next street, nor when two owls swooped overhead. In fact, it was nearly midnight before the cat moved at all.*

from the first book from J.K. Rowling's Harry Potter series [6]. The first column in the table contains the input words. The second column contains the suggested RGB values for the input words. The third column contains the keywords used to tag the suggested sound files for the input words, and the fourth column contains Narratarium's certainty in the suggested sound files (the sound file URLs and metadata have been excluded for the sake of brevity). Finally, the fourth column contains the total time to generate the color and sound guess for each input word. These results were generated using the incremental, non-backwards spreading graph algorithm with a decay rate of 0.5 (the effect of which can be seen in the fourth column, where values sometimes decrease across subsequent words by a factor of 0.5), a `maxHistory` of 10, a `branch` of 3, and a `maxDepth` of 2.

Figure 4-8 illustrates the intractability of analyzing correctness. Even if the complete metadata for each sound file were presented, it is important to realize that the metadata for any sound file F may not necessarily be the most accurate or appropriate metadata for the actual contents of F. Rather, the metadata (upon which the corpus is built, and hypotheses generated) is the information that the human who



Input Word	RGB Value	Sound File Hypothesis Keywords	Sound Certainty	Time (s)
It	(255, 188, 61)	None	None	1.07
didn't	None	set([u'corn'])	35	0.09
so	None	set([u'corn'])	35	0.00
much	None	set([u'corn'])	56.5	0.32
as	None	set([u'corn'])	56.5	0.00
quiver	None	set([u'shake'])	44.06363505	0.68
when	None	set([u'shake'])	44.06363505	0.00
a	None	set([u'shake'])	44.06363505	0.00
car	(91, 0, 255)	set([u'corn'])	1348.565166	0.92
door	(66, 172, 79)	set([u'car', u'door'])	1616.446102	0.94
slammed	(112, 136, 62)	set([u'slam', u'door'])	1546.183066	0.73
on	(112, 136, 62)	set([u'slam', u'door'])	1546.183066	0.00
the	(112, 136, 62)	set([u'slam', u'door'])	1546.183066	0.00
next	(90, 65, 114)	set([u'slam', u'door'])	773.091533	0.75
street	(0, 0, 0)	set([u'street', u'salesman'])	805.3946238	0.71
nor	(0, 0, 0)	set([u'street', u'salesman'])	805.3946238	0.00
when	(0, 0, 0)	set([u'street', u'salesman'])	805.3946238	0.00
two	None	set([u'street', u'salesman'])	402.6973119	0.60
owls	None	set([u'owl', u'scotland', u'bird', u'fieldrecor'])	1403.51231	1.03
swooped	(255, 255, 255)	set([u'owl', u'scotland', u'bird', u'fieldrecor'])	701.7561549	0.81
overhead	(58, 82, 172)	set([u'owl', u'scotland', u'bird', u'fieldrecor'])	350.8780775	0.99
in	(58, 82, 172)	set([u'owl', u'scotland', u'bird', u'fieldrecor'])	350.8780775	0.00
fact	(43, 173, 172)	set([u'owl', u'scotland', u'bird', u'fieldrecor'])	175.4390387	1.07
it	(43, 173, 172)	set([u'owl', u'scotland', u'bird', u'fieldrecor'])	175.4390387	0.00
was	(43, 173, 172)	set([u'owl', u'scotland', u'bird', u'fieldrecor'])	175.4390387	0.00
nearly	None	set([u'owl', u'scotland', u'bird', u'fieldrecor'])	87.71951937	0.87
midnight	(21, 17, 67)	set([u'noon', u'gong', u'midnight', u'clock'])	76.56607629	1.60
before	None	set([u'noon', u'gong', u'midnight', u'clock'])	38.28303815	1.17
the	None	set([u'noon', u'gong', u'midnight', u'clock'])	38.28303815	0.00
cat	(0, 0, 0)	set([u'cat'])	591	1.22
moved	(162, 0, 110)	set([u'cat'])	295.5	1.50
at	(162, 0, 110)	set([u'cat'])	295.5	0.00
all	(162, 0, 110)	set([u'cat'])	295.5	0.00

Figure 4-8: Hypotheses Generated for a Text Input Sample

uploaded F believes is the most important/descriptive of F. If another human were to upload F, the description and tags for this second submission would likely differ from the original description and tags, especially if the sound content of F is complex and involves many different sound effects.

Thus, it would be more meaningful to attempt to optimize Narratarium's sound and color corpora, *divisi*'s relatedness matrix, and Narratarium's configurable values than it would be to attempt to define a metric for accuracy.

Chapter 5

Conclusion and Future Work

Narratarium is a MIT Media Lab project and product that uses commonsense knowledge and natural language processing algorithms to enhance the storytelling experience with sounds and colors. A user can either tell a story out loud, enter a story into Narratarium using a keyboard, or select a prerecorded story, and Narratarium will project the words of the story onto the walls of the room in which it resides while changing the color of the room and playing sound effects based on the content of the story.

Narratarium utilizes a number of natural language processing algorithms, previously developed at the MIT Media Lab, so that it can not only generate colors and sounds based off the story being told, but also based off the extensive set of experience that humans possess and take for granted. It uses a commonsense knowledge base from ConceptNet's[3] large set of binary-relation assertions compiled from facts, rules, stories, and descriptions from the Open Mind Common Sense project [7], in conjunction with Divisi[8], a mathematics toolkit with special applications for reasoning over semantic networks, as the foundation of its commonsense reasoning logic. It uses a modified version of Colorizer[2] to generate color guesses, and introduces a number of new and innovative algorithms to generate sound guesses, including a naive algorithm that performs with exceptionally low latency and a number of graph-based algorithms that are highly customizable for a desired level of latency or accuracy.

Although Narratarium draws its sound effects from the Freesound corpus[1], the strategies and algorithms it employ are completely modular and can be used with any set of sound files that are tagged with keywords.

5.1 Future Work

There are many opportunities for future work on Narratarium.

From a practical standpoint, the many configurable fields in Narratarium's back end algorithms must converge to fixed values before the consumer-facing product can be delivered, and a considerable amount of fine tuning must be done to achieve convergence. In addition, the sound corpus was generated using a database dump that is nearly four years out of date. A more recent version of the Freesound sound database will assuredly be much more comprehensive and representative of the human sound experience.

From a theoretical standpoint, Narratarium's back end algorithms can be modified to exploit more correlations between words in context. For example, they can be improved to support multi-word concepts. Currently, Narratarium considers separate words as separate concepts, although many multi-word concepts exist in ConceptNet's corpus, and thus *divisi*'s relatedness matrix.

From a performance standpoint, Narratarium's color guessing method, which borrows from Colorizer's [2] algorithms, can likely be optimized further. Narratarium generates a color and a sound hypothesis for each word that the user speaks or enters. As a result, the total computation time per input word is the sum of both `SoundGuesser`'s computation time and `ColorGuesser`'s computation time. At this point, generating a new color guess can take anywhere between 0.5 and 1.0 seconds, while generating a new sound guess with reasonable accuracy takes less than 0.5 seconds. Thus, any efforts to reduce Narratarium's latency are best spent improving upon `ColorGuesser`'s efficiency.

Appendix A

Figures

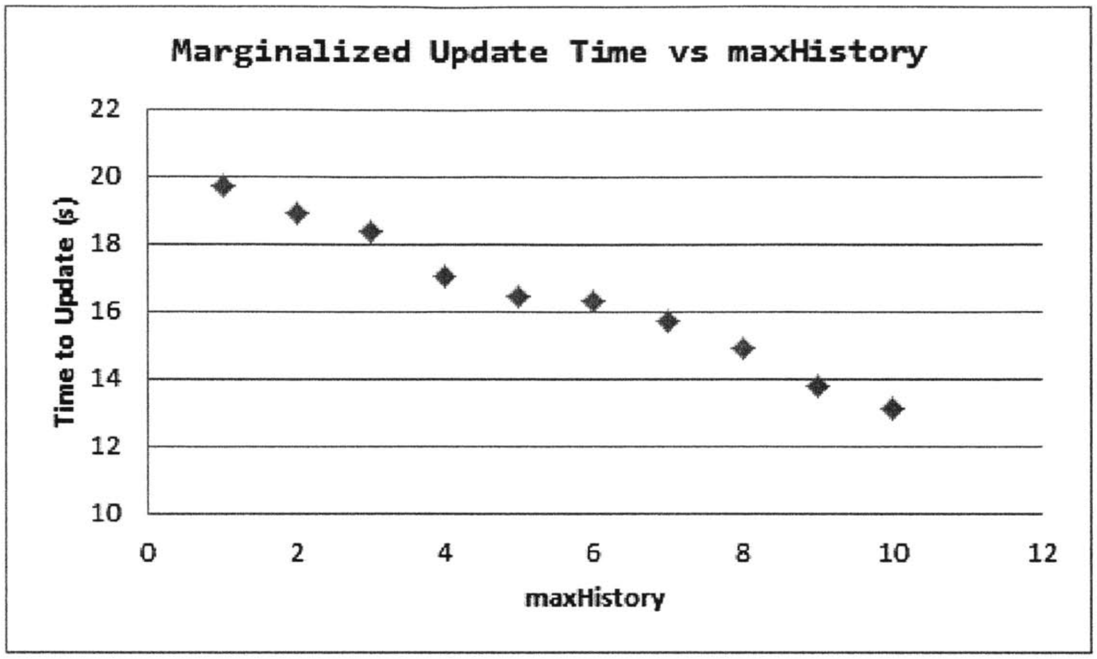


Figure A-1: Average Performance of Graph Guessing Method Variant 1 (Incremental Updates with Backwards Spreading Disallowed), Marginalized over maxHistory

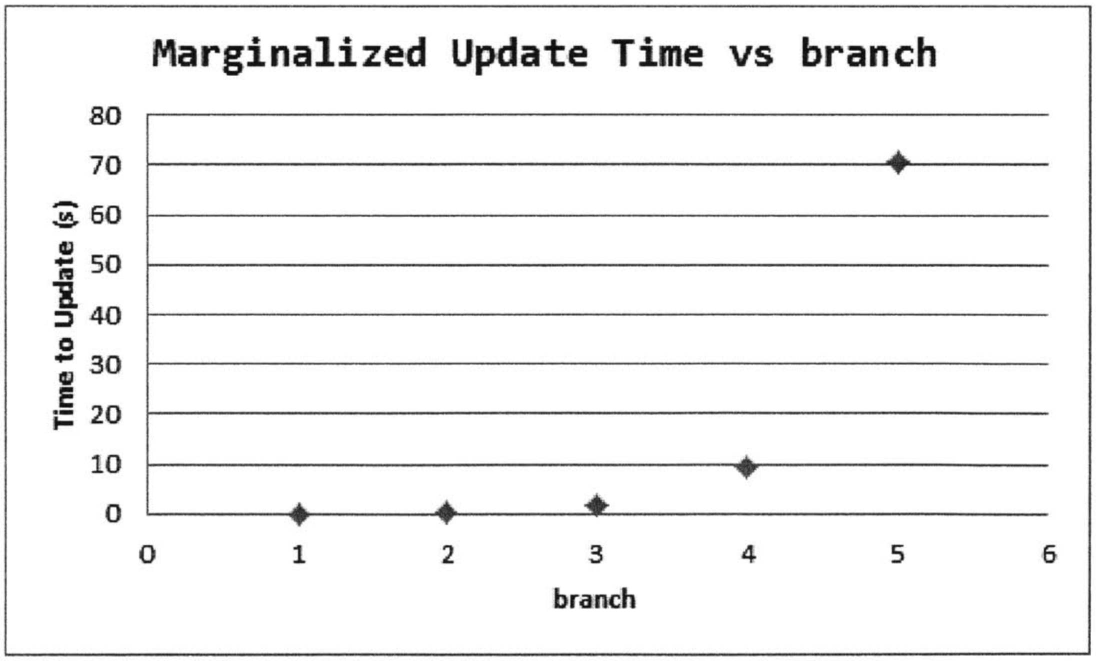


Figure A-2: Average Performance of Graph Guessing Method Variant 1 (Incremental Updates with Backwards Spreading Disallowed), Marginalized over branch

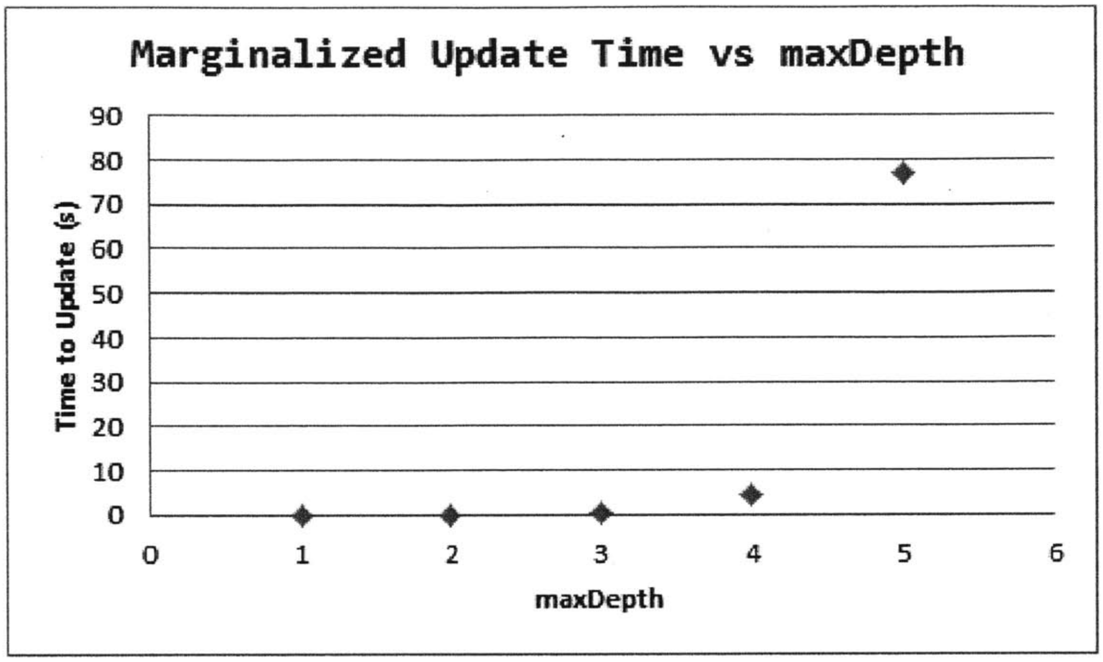


Figure A-3: Average Performance of Graph Guessing Method Variant 1 (Incremental Updates with Backwards Spreading Disallowed), Marginalized over maxDepth

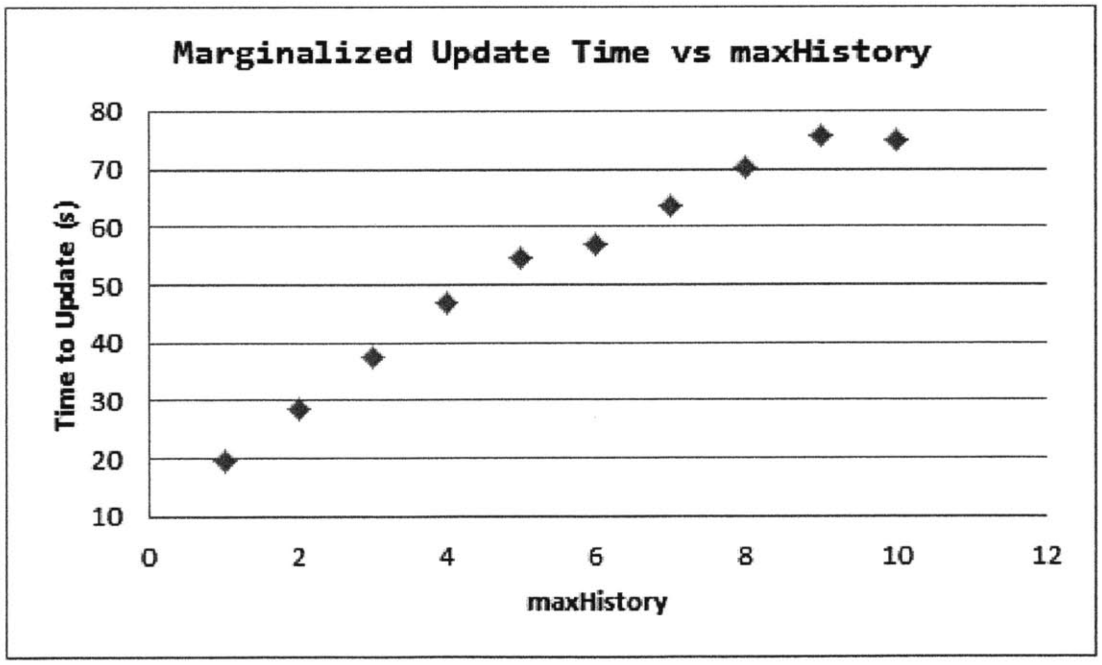


Figure A-4: Average Performance of Graph Guessing Method Variant 3 (Complete Updates with Backwards Spreading Disallowed), Marginalized over maxHistory

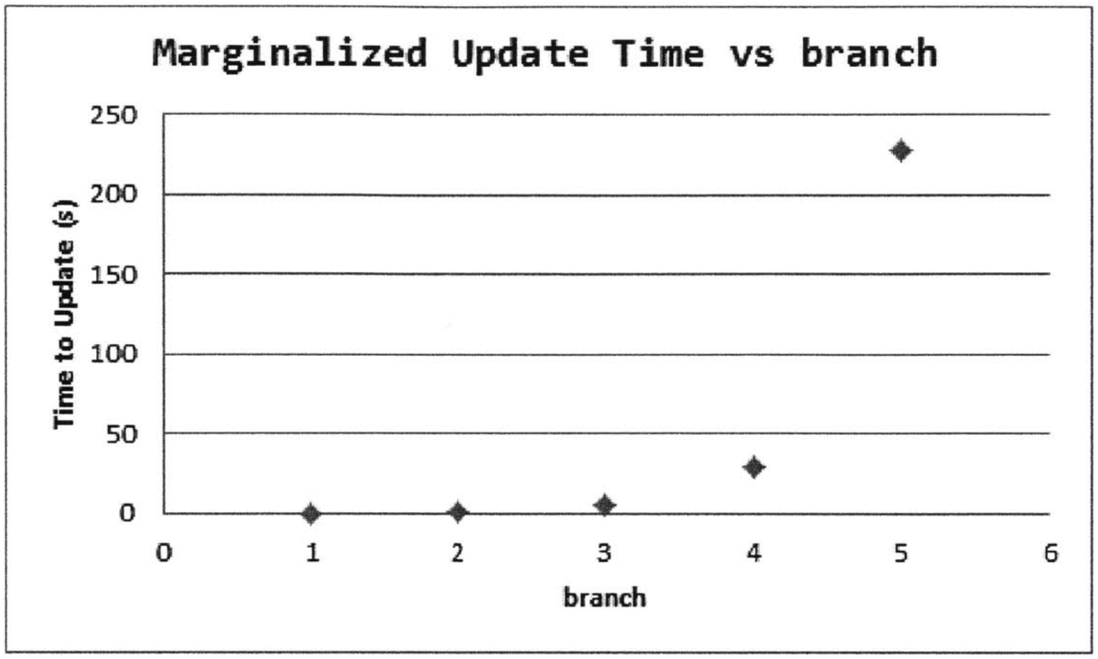


Figure A-5: Average Performance of Graph Guessing Method Variant 3 (Complete Updates with Backwards Spreading Disallowed), Marginalized over branch

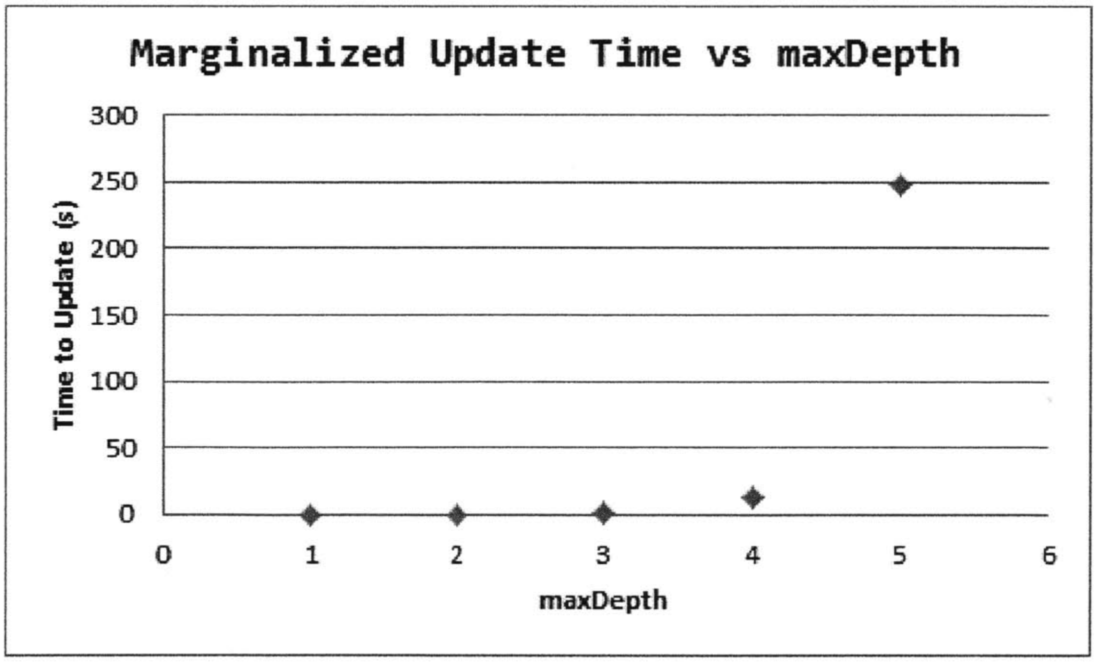


Figure A-6: Average Performance of Graph Guessing Method Variant 3 (Complete Updates with Backwards Spreading Disallowed), Marginalized over maxDepth

Bibliography

[1] Freesound, August 2009.

[2] Catherine Havasi, Robert Speer, and Justin Holmgren. Automated color selection using semantic knowledge. *Proceedings of AAAI CSK, Arlington, USA*, 2010.

[3] H Liu and P Singh. Conceptnet a practical commonsense reasoning tool-kit. *BT Technology Journal*, 22(4):211–226, 2004.

[4] Randall Munroe. Color survey results, May 2010.

[5] Jakob Nielsen. *Usability Engineering.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[6] Inc. Staff Scholastic, J.K. Rowling, M. Grandpre, and K. Kibuishi. *Harry Potter and the Sorcerer's Stone.* Harry Potter Series. Scholastic, Incorporated, 2013.

[7] Push Singh, Thomas Lin, Erik T Mueller, Grace Lim, Travell Perkins, and Wan Li Zhu. Open mind common sense: Knowledge acquisition from the general public. In *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, pages 1223–1237. Springer, 2002.

[8] Rob Speer, Kenneth Arnold, and Catherine Havasi. Divisi: Learning from semantic networks and sparse svd. In *Proc. 9th Python in Science Conf.(SCIPY 2010)*, 2010.

[9] Robert H Speer, Catherine Havasi, K Nichole Treadway, and Henry Lieberman. Finding your way in a multi-dimensional semantic space with luminoso. In *Proceedings of the 15th international conference on Intelligent user interfaces*, pages 385–388. ACM, 2010.