Learning Commonsense Knowledge from the

Interpretation of Individual Experiences

by

Sam Wyatt Glidden

S.B., M.I.T., 2008

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2009

Author_____

Department of Electrical Engineering and Computer Science
May 22, 2009

Certified by_____

Patrick Winston, Ford Professor of Artificial Intelligence and Computer Science
Thesis Supervisor

Accepted by_____

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

Learning Commonsense Knowledge from the
Interpretation of Individual Experiences
by
Sam Wyatt Glidden

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 2009

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

Abstract

To understand human intelligence, we need to discover how we learn commonsense knowledge. We humans are able to infer generalizations of knowledge, make predictions about the future, and answer questions based on what we've experienced. In this thesis I present a method for performing these commonsense tasks in an artificial intelligence system. I introduce a data type called a *chain* which clusters together similar experiences. I use graphs to store readily available historical and causal relations for experiences. The resulting memory system can handle three types of commonsense reasoning tasks. It can generalize, going from two specific examples to the knowledge that all birds can fly. It can predict, hypothesizing that since a dog likes to bark at people, it will bark when a burglar appears. And it can answer questions, providing a response when asked about the location of my car. This memory system is encoded in approximately 2,000 lines of Java.

Thesis Supervisor: Patrick Winston
Title: Ford Professor of Artificial Intelligence and Computer Science

Acknowledgements

I would like to first acknowledge Patrick Winston for his constant support and for the generosity with which he shared his experience and wisdom. His help in developing ideas, programming in Genesis, and indeed writing this thesis has been invaluable. Without his guidance I would not have found success.

I am also indebted to Michael Klein and Adam Kraft, both of who would lend an ear or an idea. Their energies in the project and for artificial intelligence in general greatly encouraged my own.

I would like to thank Matthew Peddie for random brainstorming sessions about the nature of intelligence.

Finally, I must thank my family. They always supported me in everything I did.

Table of Contents

List of Figures

Introduction

My contribution to the goal of understanding human intelligence is a strategy to learn commonsense knowledge from a collection of individual experiences. How is it that we humans can learn clear and effective commonsense knowledge from a small set of examples? How do we handle storing a lifetime of memories and access them quickly and efficiently? How to we aggregate similar experiences together to form the commonsense truths that we use every day? I propose a system for clustering together similar experiences in a manner that enables extraction of similarities and formulation of new knowledge.

Specifically, I have implemented a memory system with three capabilities, each of which was a goal of the research:

1. Extract general statements of knowledge from collections of individual experiences
2. Make predictions about the future based on current and past experiences
3. Answer questions about the present based on past experiences.

As a human, I exhibit these capabilities when I:

1. Believe birds can fly because I've seen many examples of individual birds flying (Task 1 above).
2. Expect that when a thief appears at my house, my dog will bark, because when the mailman appeared my dog barked (Task 2 above).
3. Recall immediately that my car is parked in my garage, even though I parked it there yesterday and I've had thousands of unrelated experiences since then (Task 3 above).

When we believe, expect, and recall, we perform computations that we think of as part of common sense. My thesis is to duplicate these "obvious" computations in an artificial intelligence program. In doing so, I have developed a data type I call a *chain*, which is both a cluster of similar experiences and a

single descriptive label. A small amount of computation on this label builds commonsense generalizations of knowledge, thus accomplishing goal 1. Furthermore, by combining chains with a set of graphs to record the historical order of and explicit causal relationships between experiences, I reach goals 2 and 3.

In Part 1 below, I outline my memory system and introduce necessary vocabulary. Then, in Part 2, I discuss the nature of my memory system in detail, describing how it solves the three goals listed previously. In Part 3, I explain the steps and process I took to develop my contributions. Part 4 contains a demonstration of the capabilities of the memory system and I specifically show it tackling the examples introduced in this introduction.

PART 1: A Glossary

Genesis: The Foundation

Genesis is the name of a test-bed system developed by Professor Patrick Winston and his students aimed at building an understanding of the computations that enable human intelligence. Genesis allows for the easy combination of substantial contributions from many different researchers, me included.

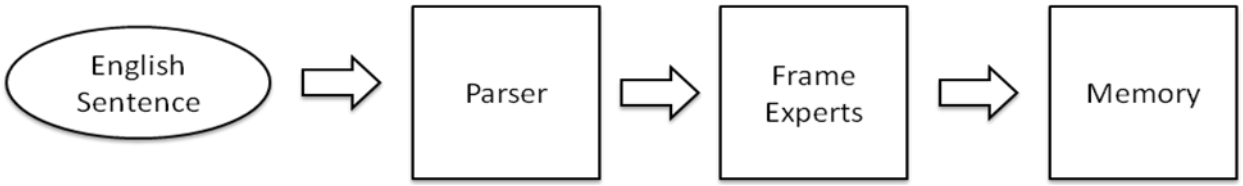


Figure 1: The flow of data in Genesis

The inputs and outputs of my system are the prescribed data types that every contributor uses. The input is a data type known as a *thing*, described in detail a later section. Each thing is a tidbit of knowledge, usually from one English sentence, represented in a specific manner. The different manners of representing information in Genesis are called *frames*.

The output of my system varies depending on the situation and the needs of the user. In situations where the memory is making generalizations from individual experiences, the output could be a true or false value telling whether or not a statement is reasonable given prior data. It could also be a set of experiences that combine to make a generalization. Whenever the memory is given new data, it may offer a prediction in the form of an instantiated frame. Finally, the answers to questions might be simple things that describe location, or provide the last known experience about a topic, or simply state yes or no.

Threads: Storing Hierarchical Knowledge

Threads are an idea presented by the A.I. researchers Greenblatt and Vaina in their 1979 work [1]. They are way of representing hierarchical knowledge in a form which is easy to understand and combine.

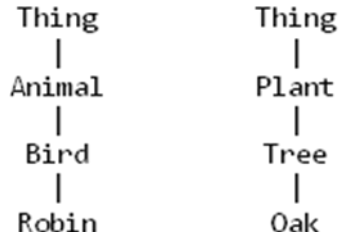


Figure 2: Examples of threads

Genesis uses threads extensively to record basic meanings of words. In Part 3 of this thesis, I will discuss threads in greater depth and outline why we choose them to use in Genesis.

Things: Core Data Type

Things are the basic data type used by the Genesis system. Each thing has a collection of threads, which is called a *bundle*. One of the threads in a bundle is specially marked as the "primed thread". The primed thread can be thought of as the basic descriptor of the thing; all other threads are additional details or secondary bits of information.

There are several classes of things:

1. *Thing*: A basic, single thing with a single bundle.
2. *Derivative*: An extension of a thing which contains another thing as a subject.
3. *Relation*: An extension of a thing which contains two other things: a subject and an object.
4. *Sequence*: An extension of a thing which contains any number of ordered things.

We can combine things without limit. For example, we can have a sequence of relations, or a derivative with a sequence as its subject. It is often useful to think of a thing collection as a tree. The

top-level thing is the top node of a tree; its children, be they derivatives, relations, sequences, or simple things, form the next level of the tree. The children's children form the next level, and so forth, until the entire structure of things is presented.

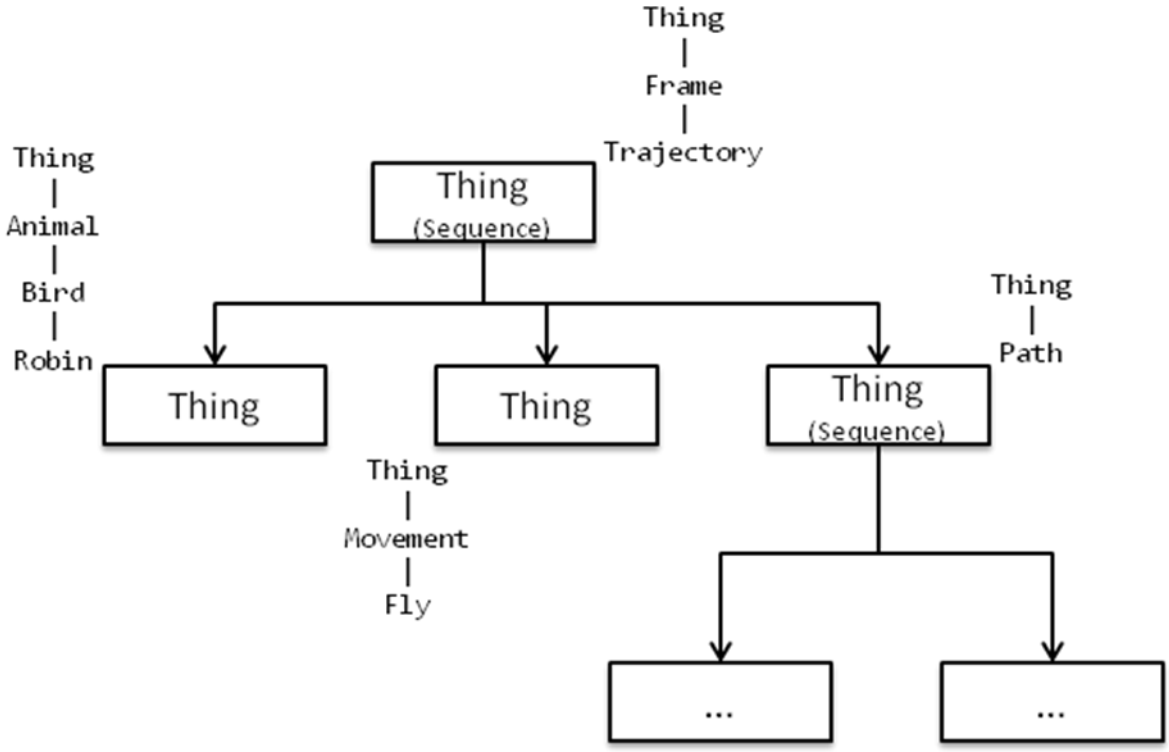


Figure 3: A view of a thing as a tree. Each node of the tree is a thing that contains a bundle of threads. The threads and the structure of the tree are varied to contain different information.

Frames: Representations Instantiated

Genesis needs to represent knowledge in a variety of ways. Each different way is called a frame, and a frame is simply a predefined structure of things (things, derivatives, relations, and sequences) stuck together in a meaningful way. For example, there is a frame to represent trajectories, where trajectories are a knowledge representation developed by Ray Jackendoff in 1983 [2]. They are good at encoding information about movement and path.

For instance, consider the sentence “A zombie hobbled to a house.” The knowledge in this sentence is easily represented as a trajectory, highlighting the mover, the means of locomotion, and the path. The sentence is used to fill in a trajectory frame.

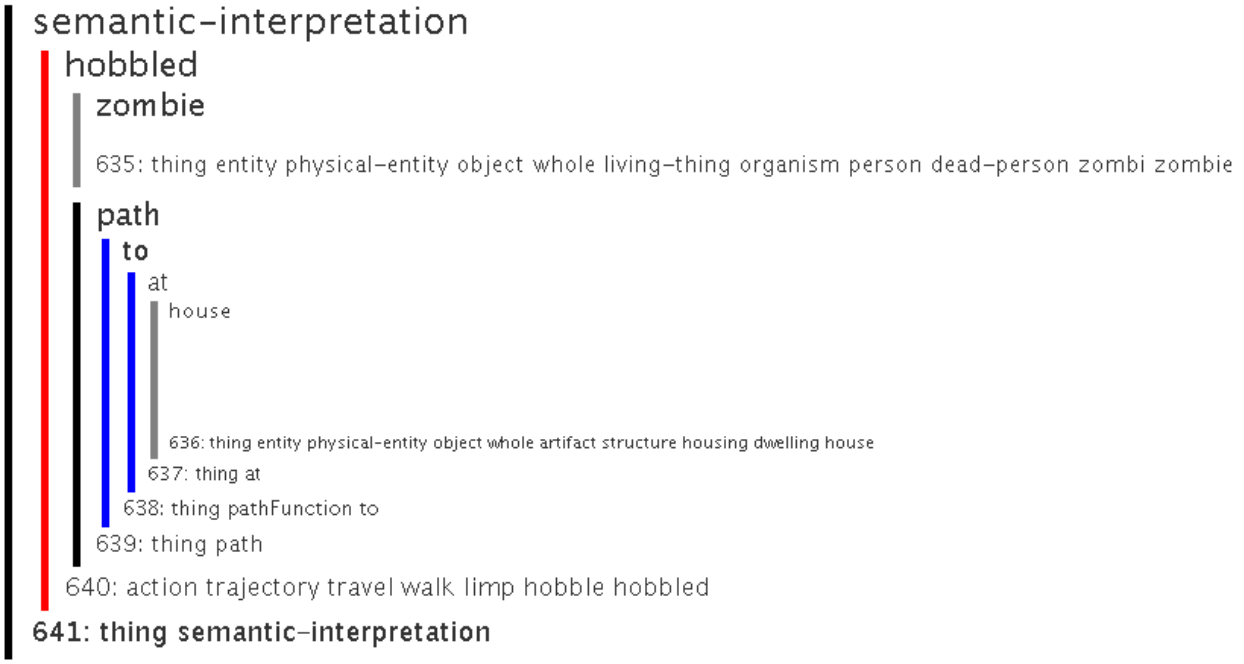


Figure 4: A trajectory frame representing the sentence "A zombie hobbled to a house". This is a more compact way of displaying a thing than presented in Figure 3, but equivalent. It is taken from a screenshot of the running Genesis program.

Genesis uses a variety of frames to encode different information, including trajectories, transitions, causes, roles, coerces, and times.

Lattice-learning: Dividing Up Data

Michael Klein [3] presented in his MIT Master’s Thesis an algorithm for classifying hierarchical data positively or negatively given a set of positive examples and a set of negative examples. His algorithm works perfectly with the threads that Genesis uses. In a nutshell, from a set of positive threads and a set of negative threads, Klein’s lattice-learning algorithm lazily divides up the space of all possible threads. Then, the algorithm can decide if any given thread falls in the domain of the positive threads or not. His

algorithm is a key mechanism for extracting generalizations from clusters of experiences. I make heavy use of it.

Near-Miss: A Guiding Principle

Patrick Winston [4] in his thesis presented an idea known as the near-miss principle. Simple in nature, it is a powerful concept that guides the fundamental operation of my memory system. The near-miss principle states that learning takes place when you receive a new piece of data very similar to what you already know. If the new data varies in only one aspect, it is easy to conclude that that aspect is the important factor in any varying classification changes. Winston describes a program that learns the definition of an arch. He provides positive examples of arches and negative examples of non-arches that fail to be classified as arches in only one aspect. This allows his program to highlight the distinguishing characteristics of an arch and develop a comprehensive description. I use his near-miss technique for learning in my memory system to decide how to cluster together similar experiences and learn from them. Details will be presented in Part 2.

PART 2: The Memory System

Overview

The central piece of my thesis is a memory system for Genesis. This memory system is responsible for storing knowledge in ways that facilitate intelligent behavior. The goals accomplished by my memory are to:

1. Infer generalizations from specific experiences
2. Make predictions about the future based on past events
3. Answer questions based on past events

It is important that the memory behaves in a manner that is biologically plausible, as the mission of the Genesis project is to understand human intelligence. As intelligent biological systems, humans are able to infer new knowledge and make conceptual leaps with a limited number of examples. A single piece of evidence is often sufficient for a person to make a confident decision. This steered my development away from statistical artificial intelligence and towards symbolic and representational reasoning. Further discussion of the process by which I developed the memory system occurs in Part 4.

My system has four main parts. These parts handle both the storage of information and intelligent operation on that information. In the course of my work, I have found that the way in which information should be stored depends highly on how it is to be used.

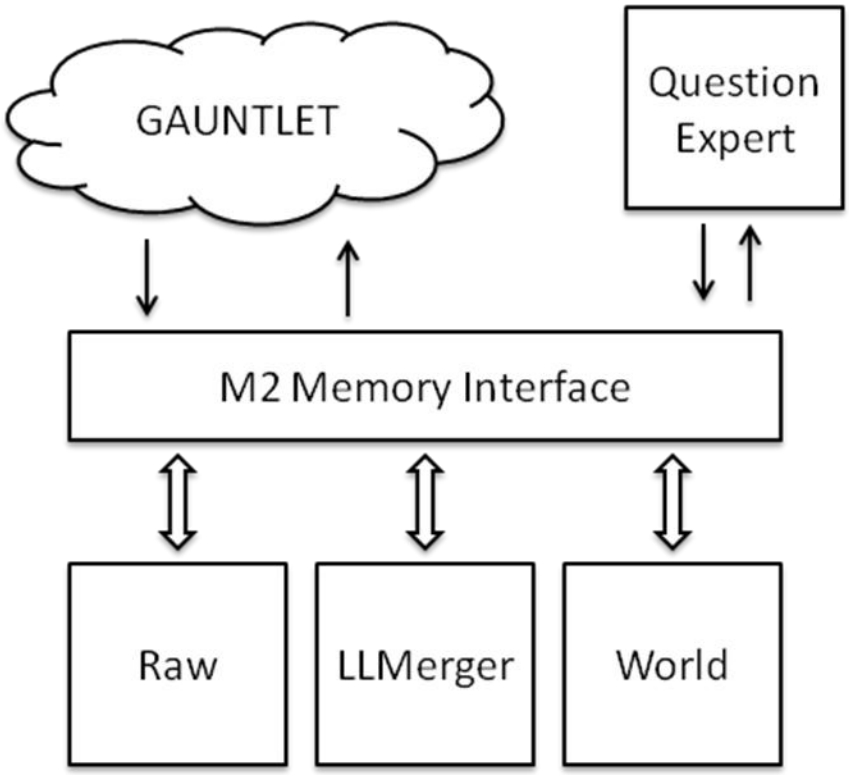


Figure 5: The structure of the memory system.

The M2 class serves as the primary interface to the memory system. M2 routes all information to the appropriate places. Connected to M2 are Raw, LLMerger, and World. Raw holds unprocessed things; LLMerger processes things using the lattice-learning algorithm; and World tracks things to maintain an idea of the state of the world.

Raw, acting as a storage bin, keeps track of each thing the memory has seen and how many times it has seen it. Functionally, Raw answers questions about the frequency of a thing or about things which talk about a given thing.

The second piece, LLMerger, is the workhorse of the memory's ability to infer generalizations from individual pieces of knowledge. LLMerger clusters similar frames together as *chains*. Chains are the data type I have developed to represent conceptually related groups of things. A chain makes it computationally efficient to store and search through many things, which becomes essential as the

memory system grows in size. The lattice-learning algorithm powers the clustering process used in chains.

Third, World is the vehicle for making predictions. World stores historical and causal graphs that relate the different frames seen by Genesis. It interacts with the chains of LLMerger to make predictions through analogies. The efficient nature of chains makes this interaction straightforward.

Finally, there is the Question Expert. The Question Expert exists as an entity separate from main memory and interacts directly only with M2. This separation exists because the Question Expert doesn't need to store information—it only calls upon data the memory proper has already organized. The Question Expert consists of a set of strategies for answering questions and the associated queries to memory necessary to formulate answers.

Distance Metrics

I need to collect similar data together in order to infer generalizations, make predictions, and answer questions. I have developed distance metrics as a way to decide how similar two items are. In the final iteration of my work, these distance metrics proved useful in limited situations; however, I discovered that by storing things inside the chain data type I developed, I could achieve the same goals that distance metrics reach without as much raw computation. The distance metrics outlined below are still used to rank small collections of things and threads, but do not drive the main functions of the memory.

Thread-Distance

Because threads have simple structure, it is easy to make a metric describing how similar two threads are. My procedure is:

1. Find the *edit distance* between the two threads, where edit distance is defined as the minimum number of changes (additions, subtractions, or substitutions) needed to make one thread equal to the other.
2. Divide this distance by the length of the longerthread in order to normalize it.

Thus, the distance between a thread talking about a robin and a thread talking about a bluebird would be small, because both those threads share many common elements. However, the distance between a robin thread and a tugboat thread would be long—robins and tugboats share very few similar classifications.

Thing-Distance

I also developed a metric for finding the distance between two things. Because each thing could actually be any of a relation, a derivate, or a sequence, this can be a complicated procedure. Every thing can be thought of as a tree. Thus, a comparison of things is a comparison of two trees, where the order of the nodes in the tree can matter. Each node on the tree is a thing, which contains a bundle of threads. I use an algorithm called Needleman-Wunsch [5] to optimally pair up the ordered children things of two things. I then use a procedure that finds the distance between the things' bundles by comparing the two bundles' threads in the most favorable manner. As such I walk down the two trees from top to bottom, weighing things at the top more highly than at the bottom. This choice was made because it works well in practice. First, it magnifies the distance between different types of frames. Secondly, as a general rule of thumb, the more important representational elements in a frame are higher up in the tree structure. Thus, they are weighted more heavily.

Shortcomings

These two metrics (thing and thread distances) provide a way to compare and cluster similar frames together. This is an important step in making generalizations. Intuitively, if two things are close in thing-

distance, they should be clustered together. Unfortunately, a thing-distance calculation is very expensive. Therefore, I have developed the data type known as a chain to contain clusters of similar things. Chains can be thought of as “labels” for clusters of things. From this label alone I can decided if a new thing belongs in the cluster.

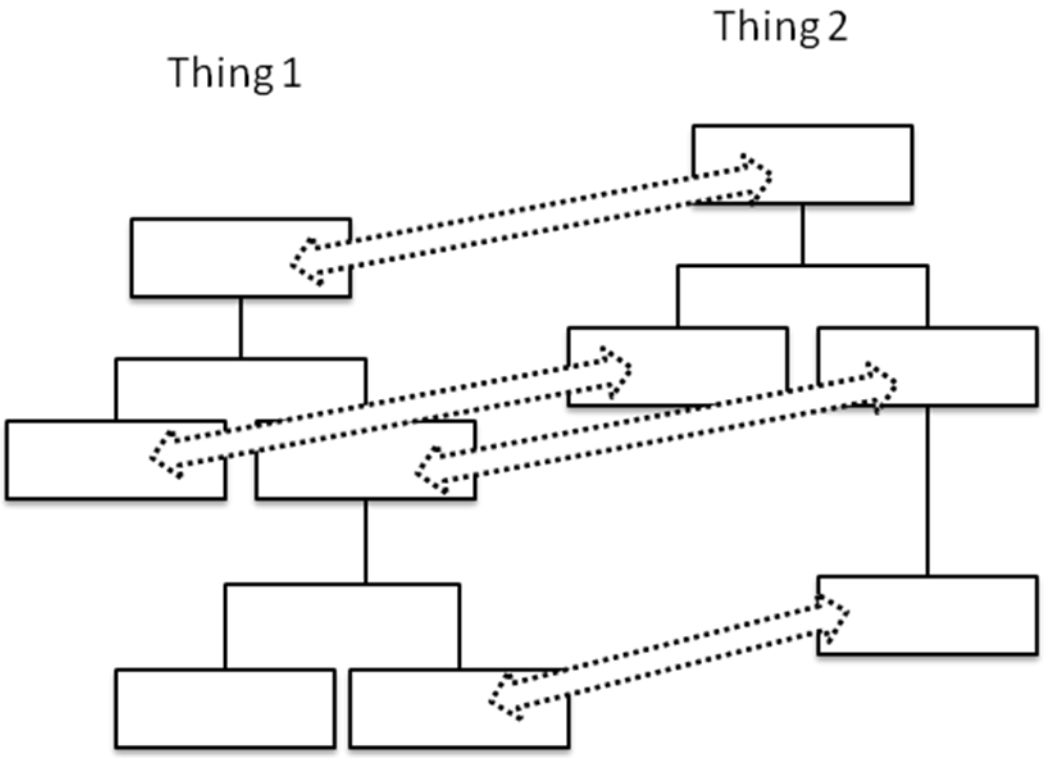


Figure 6: Comparing two things involves comparing their components. I optimally match up things by level to generate a consistent and optimal distance value.

Chains: Clusters of Things

Chains are the core data type for organizing and storing knowledge. They are responsible for making it easy to create generalizations, prediction, and answers, and are central to my system and my thesis.

A chain is designed to summarize the things it contains. It is the result of flattening things and lining up their similar elements. This flattened view is conceptually and computationally simple. It is easy to determine whether or not a thing belongs in a certain chain by looking at this flattened view.

Figure 7: An example of a chain. Note that the second cell has two elements: both are positive examples. This image is taken from a screenshot of Genesis.

As you can see, a chain is made up of cells. Each cell corresponds to the bundles of threads in the things the chain describe. Each cell has two sets of threads: a positive example set and a negative example set. These two sets form the input to the lattice-learning algorithm described earlier. Thus, I can evaluate whether or not a new thing falls inside the chain by running lattice-learning on each cell.

There are a number of operations that can be performed with chains:

- Generalizing knowledge
- Creating neighborhoods of things
- Creating a chain
- Clarifying a chain with new positive and negative examples
- Splitting a chain as the result of conflicting examples
- Merging chains

A new chain is created from a single thing. I enlarge chains by adding matching input things. This creates a collection of similar things. Sometimes, Genesis receives input that conflicts with the knowledge contained in a chain. For example, a chain could describe a dog flying to a tree. If I tell Genesis that dogs cannot fly, it is necessary to reconstruct that chain to include that negative example knowledge. This consists of splitting a single chain into multiple smaller chains. Inversely, I may discover that two chains have continuous overlap in knowledge—it then becomes appropriate to merge the chains.

Making Generalizations: The First Task of the Memory

It may already be apparent how chains can be used to make generalizations from specific chunks of knowledge. A chain represents more than the individual things inside it. By using the lattice-learning algorithm, I can infer all the things which would belong to a single chain. Consider, for instance, a basic chain formed from two things: one stating that a robin flew to a tree, the other that a dog cannot fly at all.

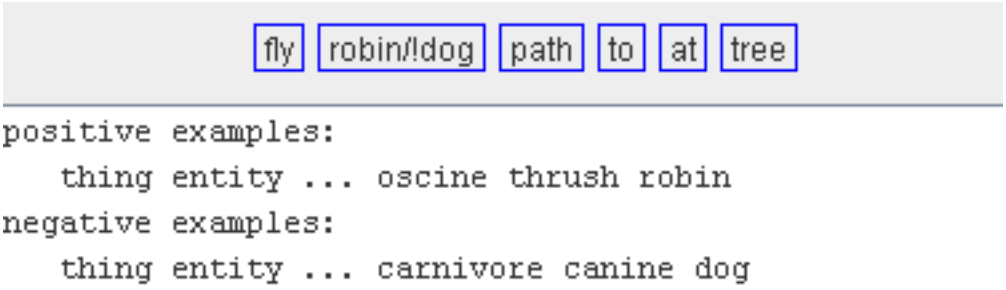


Figure 8: A chain can be used to generalize experiences by running lattice-learning on each cell. The positive and negative example threads seen here are for the second cell. Lattice-learning uses them to generalize this chain to include all birds but reject mammals.

The meaning of this chain is far more encompassing. From the two example things (the robin and dog trajectories), the knowledge encoded in the Genesis's threads, and the lattice-learning algorithm, we can hypothesize that anything similar to a robin, but not like a dog, can fly to a tree. Thus, Genesis has the ability to assess the possibility of a third statement based only on two it has seen. The assessment is similar to what we as humans would make: if I consider a trajectory very similar to the ones I've already seen, then I would assume that it is possible. Both Genesis and I would find the trajectory of a bluebird flying to a tree quite palatable.

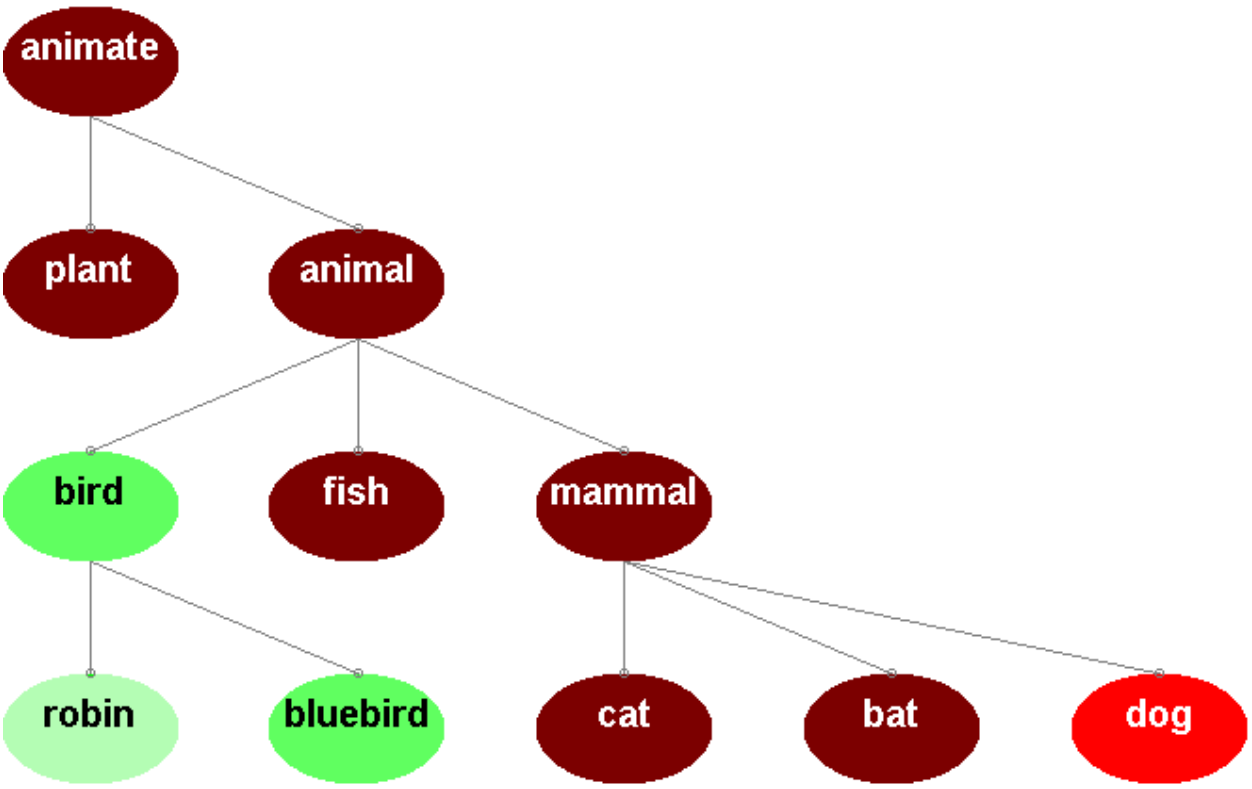


Figure 9: Lattice-learning has divided up the space of things. It has classified a bluebird positively with the robin. This image is from Patrick Winston's teaching material [6].

Matching a Thing to a Chain

Mechanically, how does Genesis decide if a thing matches a chain? This is, of course, the method by which my memory makes and uses generalizations. As a chain is a cluster of things, it is necessary only to determine whether or not a given thing belongs in that chain. As outlined earlier, the procedure for this involves looking at the cells of the chain and using lattice-learning. To see if a chain contains a new thing, I:

1. Flatten the thing, so that each of the bundles in the thing line up with a cell in the chain.
2. For the first cell in the chain, run lattice-learning, using the cell's positive and negative sets of example threads as inputs.

3. Check if the primed thread from the first bundle of the new thing is accepted by the lattice-learning algorithm.
4. Repeat for each cell-bundle pair.

If every primed thread from every bundle is accepted by lattice-learning, then the thing belongs in this chain.

Near-Misses between Things and Chains

Above, I described how a thing is matched to a chain. This matching process can produce a score for the number of "misses" a thing has with a chain. If the thing matches perfectly, it has zero misses. If it similar to a chain but varies at one single cell, it has one miss. It is very important to identify these near-misses because this is where learning takes place. Things that are a near-miss of a chain are added to the chain in order to modify the scope of things that the chain matches. Of course, things with no misses are also added to a chain—but they do not change the comprehensiveness of the chain's scope.

In order to determine whether or not a thing is a miss from a chain, I run the algorithm described above to match the thing to the chain. As the algorithm runs, I keep track of any cell where lattice-learning fails to match the new thing. If the failure to match is a "soft" miss, that is, if it implicitly failed to match rather than explicitly, I note this. If there is only one such soft miss for the entire chain, then the thing is a near-miss of the chain. In other words, a soft miss means that lattice-learning did not feel the thing's thread fell in the scope of the positive threads in the corresponding cell of the chain—but, the thread was also not explicitly forbidden by any negative example. If the thread is explicitly forbidden, then I consider this a "hard" miss and do not count the thing as a near-miss to the chain.

Using Chains to define Neighborhoods of Things

The construction of chains and the concept of near-misses let me define neighborhoods of things. This is very useful—neighborhoods of things are groups of similar things, and it is a frequent task of the memory to use one piece of knowledge to access other similar pieces.

There are two classes of neighborhoods: near-neighbors and neighbors. The near-neighbor class includes all the things that are in the same chain. As a chain is a cluster of things, that cluster is a set of near-neighbors. Thus, any thing that is in that cluster (or would belong in that cluster, as decided by the previous thing-to-chain matching algorithm) has the other things in the cluster as its near-neighbors.

The neighbor class is wider in scope. It includes all near-neighbors, plus all the things in all the chains that are one or two misses. So, given an input thing, I can find all its neighbors by aggregating the things in any chains zero, one or two misses away from the thing. The reason for accepting up to two misses is that generally chains that are near-misses of each other are merged together. In order to be effectively distinct from the near-neighbor class, the neighbor class considers all things out to two misses. Being able to summon a set of neighbors of a thing is useful for analogical reasoning and leaps of logic. The technique is used in Genesis to disambiguate words, make analogical predictions, and answer certain types of questions.

Consider the following example, illustrated in figure 10. I have a thing stating: “A dog ran to a house”. That thing is in the same chain as the statement: “A cat ran to a house”. As they are in the same chain, the cat statement is a near-neighbor of the dog statement. Now say I have another statement, saying, “A dinosaur ran to a tree”. It is in a different chain that is only two misses from the dog chain. Thus, it is a neighbor to the dog statement. Finally, I have a third chain, containing the thing: “A bird flew to the tree.” There are three misses from this chain to the dog chain, and therefore they are not neighbors at all.

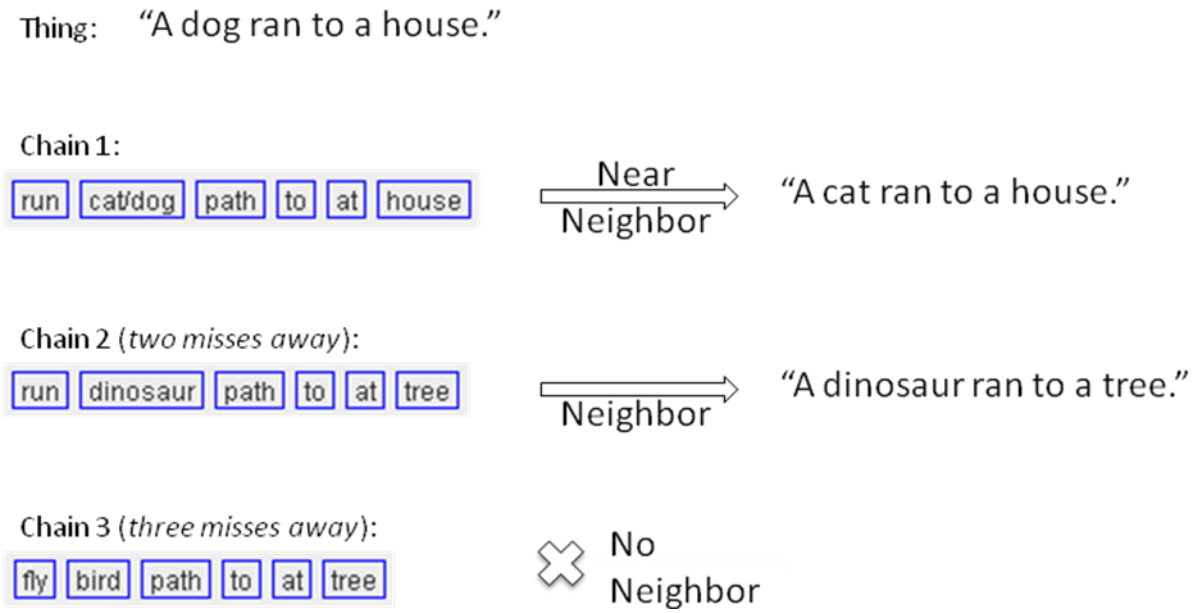


Figure 10: Example near-neighbor and neighbor to "A dog ran to a house."

Creating New Chains

Now that it is clear that chains can be used to make generalizations, I will describe how they are created and maintained. Creating a chain is simple. If I see a thing that does not belong in any existing chain, I flatten it and use its elements as a starting point for each cell in a new chain. The primed thread from each bundle in the thing forms a single example thread in the new chain's cells.

Flattening a thing is a straightforward depth-first search through the tree structure of the thing. Each time we see a bundle, we add it to our list, until we have fully explored the thing and all its children. As things have ordered children things, the search is stable, and we will consistently line up the bundles within things to the cells within chains.

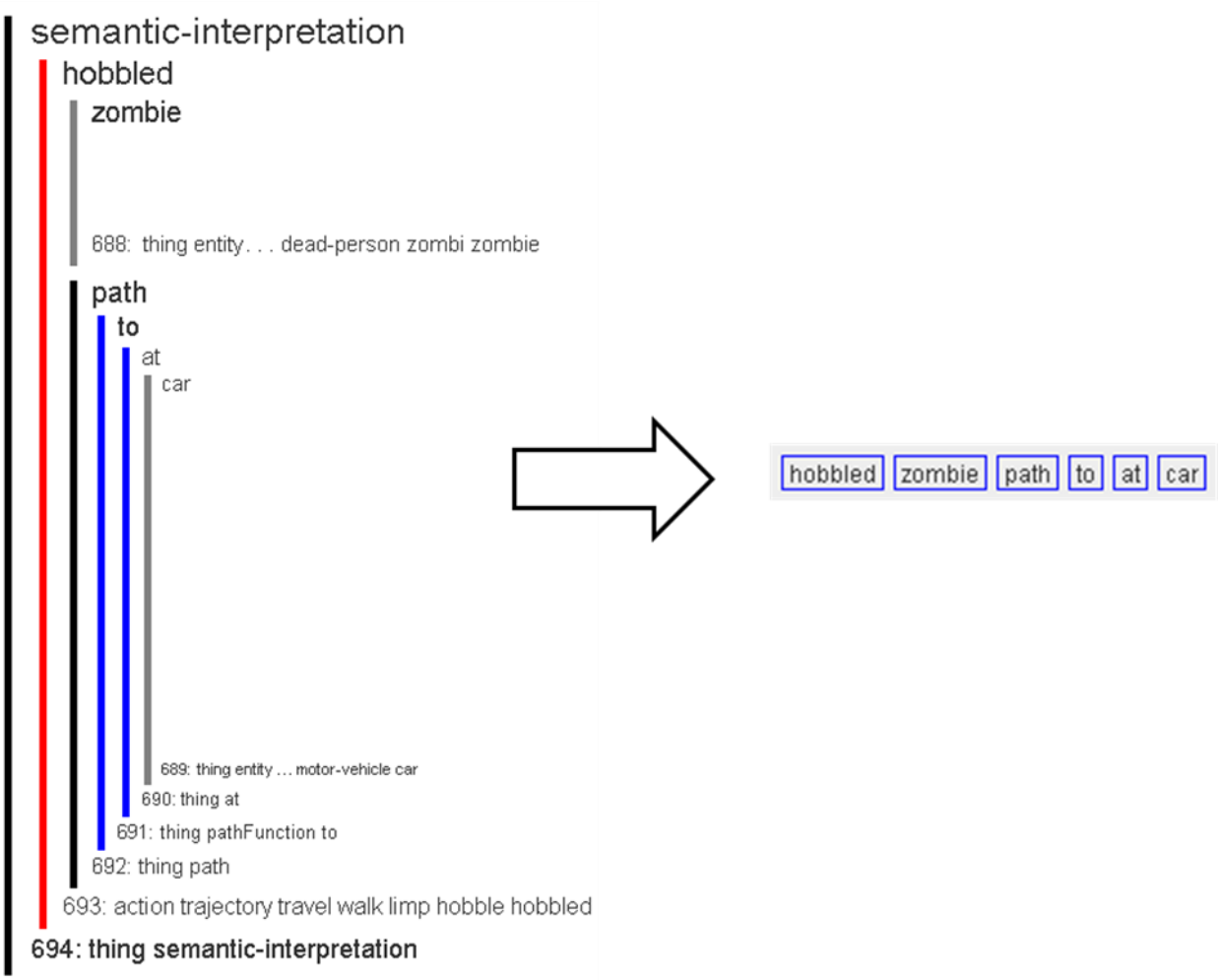


Figure 11: Creating a chain from a thing. The thing is flattened, retaining threads (not shown here).

Clarifying a Chain by Adding Positive and Negative Examples

A chain begins its existence from a single *thing* as described above. When new things are added to the memory I run the thing-to-chain matching algorithm and combine the things with matching chains of zero or one misses. Often, this addition is simple: First, I flatten the thing, so that its bundles line up appropriately with the cells of the chain. Then, I add the primed thread from each bundle to the corresponding set of positive or negative example threads in each cell. This will update the way lattice-learning operates on the chain, thus assuring that an updated set of things will match to it. Thus, the chain (acting as a cluster of things) clarifies its scope to encompass a different set of things. I also save a

copy of the original input thing. Anytime we reform the cluster, we need to refer back to the original input data. This happens when we merge clusters or when we add negative examples.

Sometimes, however, the new piece of input conflicts with a chain already in memory. This conflict is best illustrated by example.

Say that I add these three statements to the memory:

1. A dog ran to a tree.
2. A bird ran to a tree.
3. A bird flew to a tree.

The result is a chain that would claim both a bird and a dog can fly or run to the tree. From Genesis's perspective, this is logical—we've established through the input of things that birds and dogs are similar, and we know that a bird can fly to a tree. In the absence of any further information, Genesis assumes that a dog could fly as well.

Let us provide that further information. Say I add the sentence:

4. A dog cannot fly.

This is now a *negative* example to a chain I've got in memory. It says that the current chain implies something that isn't true. At this point, I need to reform the chain to avoid concluding the erroneous idea that a dog can fly to a tree—I do this by chopping the incorrect chain up and rebuilding it into several smaller chains.

The algorithm to go from a conflicting chain to several deconflicted chains works like this:

1. Delete the conflicting chain.
2. Collect all the things that went into the chain into a list.
3. Create a new chain from the first thing.

4. Attempt to merge in another thing from the list. Check to see if that merge results in any conflicts with any of the other things.
    a. IF NO CONFLICT: repeat on the next thing
    b. IF CONFLICT: do no perform the merge; instead, create a new chain from the merging thing. For here on, try to merge additional things from the list with this chain as well as the first one.
5. Once the things are combined as much as possible without creating conflicts, add the resulting chains to memory.

For the above example, the "dog cannot fly" statement causes the chain to be reformed into three chains. The first says: "a dog and a bird can run to a tree". The second says that "a bird can fly and run to a tree". The third says that "a dog cannot fly". These three chains now draw consistent and correct conclusions from the inputs provided.

Merging Chains

Over the course of running the system, chains are created and grown through the addition of knowledge encoded in things. Occasionally a new thing will match two chains. When this happens, I can conclude that those two chains conceptually overlap and should in fact be a single chain. I can merge the two chains together. It is possible to detect when two chains should be merged at the time when a thing is added to the memory, or after all conflicts have be resolved and the new things completely processed. I have chosen the wait until new things are processed to merge chains together, as this approach is more straightforward.

The chain merging algorithm is simple. I start by cloning the first chain and then I add, one at a time, the things from the second chain. This assures that each cell of the chain is properly formed. The only caveat is to be wary of creating a new chain that conflicts with something already in memory. This

situation is avoided the obvious way: do not do a merge if it results in a chain that does not make sense. I simply check a potential new chain for conflicts just as I do when I grow a chain through the addition of new things. If the new version of the chain doesn't conflict, it is added to memory; otherwise the merge is not done.

Making Predictions

By developing chains, I have provided a method for accomplishing the first task of commonsense reasoning: inferring generalizations. The second task is to make predictions about the future based on current and past events. This is a hallmark of intelligence: foreseeing challenges and potential solutions before they occur. In Genesis everything is recorded as thing, so that is format I use to make predictions. Given a current thing, I look at past things and decide if anything is expected to happen. If so, Genesis volunteers a prediction in the form of a thing describing the likely future event.

Predictions are made by the World module of the memory, as described earlier. This module both communicates with the chains stored in LLMerger and with its own internal graphs of frames. Using these collections of memories, World makes four different kinds of predictions, each with a different logical source. They are:

1. *Explicit predictions*. These are predictions developed explicitly from cause frames seen earlier. For example, if a cause frame tells the memory that when a zombie appears, the dog barks, the memory will easily predict that the next time a zombie appears the dog will bark.

2. *Analogical predictions*. These predictions are made from simple analogies with explicit cause frames. Say that "a bird flew to a man because a man appeared" is explicitly given in a cause frame. If I then present the information that a tree appeared, the memory will make the analogous substitution and predict that a bird flew to a tree.

3. *Historical predictions*. These are predictions based purely on historical precedence. If in the past the thing of a bird flying preceded the thing of a bird landing, then next time a bird flies, the memory will predict it will land.

4. *Circumstantial predictions*. These are predictions based on only the concrete objects in a thing. Historical predictions do not require a relationship between one thing and another other than temporal precedence. Circumstantial predictions look for the next thing that talks about the same concrete object the current thing talks about. For example, given a thing that talks about a zombie, the prediction will be the next frame to talk about a zombie.

There are graphs within World that stores connections between frames. These graphs record:

1. A timeline of frames that come into the memory
2. A storyline for concrete objects; for example, all the things, ordered in time, discussing a bird.
3. A network of causal relations, extracted from cause frames.

These graphs, coupled with the chains in LLMerger, enable me to make the types of predictions discussed above.

There are obvious shortcomings with each of these four types of predictions. It is no trouble to imagine a scenario where Genesis makes an unrealistic prediction about the future through one or more of these mechanisms. However, it is also possible to imagine fixing this problem with a system that analyzes both the possibility and likelihood of a prediction. One easy check is to run the prediction through the LLMerger, asking whether or not it seems plausible given current chains. This eliminates immediately any statements that we have concrete negative examples against: for instance, Genesis could eliminate any predictions that involve flying penguins as soon as it knows that penguins cannot fly.

This is an additional example of how my memory system makes commonsense reasoning straightforward. A logical extension to this thesis is a tool to analyze of the plausibility of predictions.

Answering Questions

Lastly, the memory is constructed with the task of answering questions. Rather than present intricate procedures to answer all types of questions (which is outside the scope of this thesis), I will instead outline a system of strategies which can be targeted at the appropriate types of questions. My plan, then, is this: first, identify knowledge to be extracted from the memory system and call the extraction procedures *strategies*; second, classify questions in terms of the appropriate strategies that answer them. I have developed a rudimentary system to demonstrate this in operation. My goal was to demonstrate that the memory system is organized in a way that makes answering commonsense questions simple.

As hoped, my memory already provides answer to many types of questions. Chains cluster together similar things and make it easy to assess whether or not a statement is plausible. They also make it easy to get related information regarding a topic. The graphs in World that power predictions contain the historical knowledge to make it easy to answer questions about the current and past state of the world. Thus, through careful design of the memory system, much of my work is already done.

I have developed the following demo strategies, which do little more than query the memory.

1. *Existence*: Does a frame exist explicitly in memory?
2. *Supertype*: What is the supertype of a given thing, according to the threads Genesis has?
3. *Location*: Where does the most recent trajectory frame place a thing?

When Genesis is given a question, it parses into a question frame and sends it to the memory system. The memory system analyzes the frame and decides which strategy is appropriate for answering the question. It then runs the strategy and returns the generated answer.

The strategies listed are merely a sampling of the possible ones that could be developed. The point is that the memory system makes it efficient and straightforward to extract a wide range of information. Question answering is therefore a problem of matching an English language query to the appropriate strategy, finding the answer, and outputting it back in English.

A Final View

I have created a memory system for Genesis with three goals in mind: making generalizations, predictions, and answers to questions. My solution is to cluster things together into chains. This clustering, combined with the hierarchical knowledge mined from Genesis's threads, neatly solves the generalizations problem. Extending my memory system further with graphs of historical and causal knowledge allows Genesis to make a varied set of predictions, including through the use of analogy. Finally, these memory structures can be leveraged by a set of predefined strategies to answer many different types of commonsense questions.

PART 3: My Process

The production of this thesis touched upon many ideas and technologies in artificial intelligence. My early work on a commonsense memory system focused on the different ways of representing knowledge and experiences. I tried to store experiences in a data type known as a self-organizing map. Over time additional requirements called for new and more complex features to be written into the memory code. Eventually it became difficult to recognize the original inspirational ideas. Growth and performance motivations forced me to rethink the entire structure, and using new ideas from other researchers I built the memory system that currently exists. In this section of my thesis I will address the process which resulted in the current memory system, in order to better assess its performance and feasibility as a mechanism for generating commonsense knowledge.

From the beginning Genesis used things and threads as primitive data types. My memory needed to work with those structures and it is crucial to understand them. All storage and commonsense reasoning facilities would speak in a language of things and threads. As mentioned in Part 1, those structures are organized as the predefined frames that are best at representing the knowledge contained within the given English sentence. First, I will discuss the different types of frames that Genesis uses.

Frames: Storing Knowledge

I have been referring to the ways of representing bits of knowledge or experiences that the memory operates on somewhat vaguely as frames. At this point I will make the idea of a frame concrete. There are a number of different frames which Genesis uses, each with a specific and proper definition. Remember, a frame is an abstract way of representing an experience or a bit of knowledge; the language a frame is written in are things and threads.

Three key frames that Genesis uses are:

1. Trajectories
2. Transitions
3. Causes

Each frame has its own purpose and encodes a specific niche of knowledge.

Trajectories

Trajectories are a very useful frame. Ray Jackendoff developed the trajectory representation in his 1983 work, *Semantics and Cognition* [2]. Jackendoff hypothesized that many events can be described as a trajectory, either concretely or abstractly. This matches an intuitive hypothesis of how the mind works: we think with our imaginations, visualizing physical scenes to understand them. A trajectory representation can be thought of as just such visualization.

Jackendoff noted that prepositional phrases can be used to describe paths or places. Consider the phrases: “under the car” or “to the mall”—both are obvious components of a trajectory. A full sentence such as “the mouse ran under the car” is easy to identify as a trajectory and is simply inserted into one of our trajectory frames. Other types of sentences are merely veiled trajectories—uncovering them yields usable knowledge. The sentence “The man struck the zombie with a shovel” implies the trajectory “The shovel swung from the man to the zombie”. Explicitly realizing this trajectory is essential to imagining the scene and predicting its outcome. By extracting trajectories from non-trajectories, we often gain reasoning power. Trajectories are a common type of frame stored in my memory.

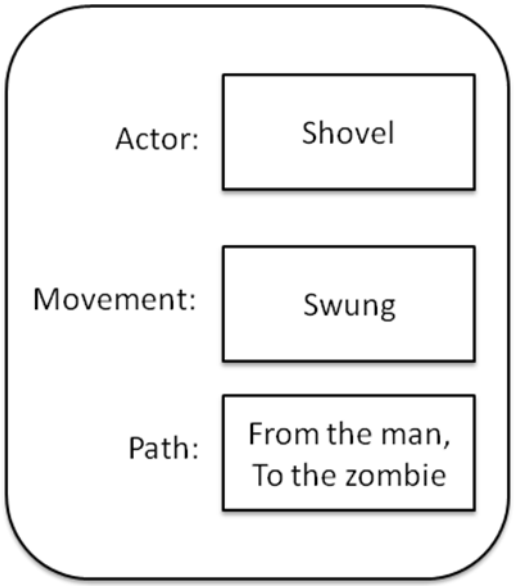


Figure 12: A trajectory frame describing the sentence: "A man struck a zombie with a shovel."

Transitions

Transitions are another useful representation, particularly in the domain of describing sequences of events or stories. Gary Borchardt identified ten transitions types: appear, disappear, change, increase, decrease, and the negation of each [7]. These transition types are coupled with a quantity. For instance, two transitions are “contact appeared” or “the distance between me and my car decreased”. Thus, a transition describes change. Some English sentences are already in transition form; others need to be converted. Consider the statement: “The fire spread”. This could be represented as the transition “the size of the fire increased”.

The real power of transitions comes from combining several together into a structure Borchardt refers to as a *transition space*. A transition space describes a series of events to form a primitive story. From this story, you can sometimes extract causality or correlation.

Consider the short story: “A man walked from his house to mall. He purchased a toy. But when he returned home, he discovered he had lost it somewhere.” Figure 13 describes that story as a transition space.

	Time 1	*Time 2*	*Time 3*	*Time 4*
Distance between the man and his house	Increased	No change	No change	Disappeared
Distance between the man and the mall	Decreased	Disappeared	No change	Appeared
A toy	No change	No change	Appeared	Disappeared

Figure 13: A transition space.

The above transition space can be used for reasoning purposes. We could hypothesize an inverse relationship between the distance from the man to his house and the distance from the man to the mall. Although not applicable for this story, we could try to infer causation if one quantity’s change preceded another.

Causes

A cause frame stores causal relations between two pieces of knowledge. In a way, it can be thought of as a meta-frame, but in fact any frame could be divided into smaller component frames. A cause frame contains two frames, where one is labeled as *causing* the other. Thus, a cause frame might say that “The dog ran to the man because the man appeared.” Here we can see that there are really two other frames in the sentence as well: a trajectory of a dog running to a man and a transition of a man appearing. The cause frame links those two frames together.

Cause frames are very important for helping the memory make predictions. If Genesis later experiences the man appearing, it can predict that the dog will run to it again. By way of analogy described in Part 2, Genesis could also predict that if a woman appeared, the dog would run to her as well. Without cause frames, Genesis would only be able to make predictions by noting co-occurring events, rather than generating them from explicit statements.

Other Frames

Genesis can handle many other kinds of frames—I've highlighted three that are common or that provide generally important information. For instance, Genesis has a frame for a path, which of course is a building block for trajectory frames. Genesis also has frames for ideas related to roles, times, coercions, places, and beliefs, and more are added all the time.

The Language of Frames: Things and Threads

I've outlined both things and threads in Part 1 of this thesis. A thing is very simple in implementation but powerful in terms of its ability to describe frames. A thing can be any combination of other things, derivatives, relations, and sequences without limit, thus encoding an arbitrarily complex frame. In additional to the mere structure of the things that make up a frame, information is also stored in the threads contained in the bundle of each and every thing.

We have populated a database of threads from WordNet, an online language repository that contains hierarchical knowledge [8]. It provides the factual information that humans take years to learn—knowledge such that a dog is an animal or that a man is a person. By using this repository of knowledge, formatted as threads, Genesis can dive into deeper semantic understanding.

There are several reasons why threads are a good idea, which Vaina and Greenblatt outline in their paper [1]. First, threads make some amount of biologically sense—Vaina and Greenblatt drew their conclusions from research about aphasia, a disorder effecting language use. A thread model for storing knowledge would explain the behavior of humans affected by aphasia. Secondly, threads as a data type for storing categorical knowledge are generally superior to the usual option of using trees. Threads are faster than trees to traverse, being a linear rather than branching structure. Of course, a full search of every thread about a topic would be equivalent to a searching a tree—the advantage comes when you can limit the search domain. Most importantly, threads are able to handle contradictions quite well.

There is nothing wrong with storing two threads that have conflicting information side-by-side; it is merely necessary that the program using the threads has some way of resolving contradictions. On the other hand, it might not even be possible to make a tree structure that conflicts with itself. For these reasons, Genesis stores categorical knowledge in threads. Thus, when Genesis talks about a "cat", it is really referring to the appropriate bundle of threads which contain the appropriate sense of the word "cat". Other threads, referring to other senses of the word, are not included—and this division is easy to make with threads, but would much harder if we stored knowledge in tree-like structures.

Parsing Sentences: Genesis

Genesis converts English-language sentences into the things that are the input to my memory. It does this through a multistep process. Genesis first uses a natural language parser to convert the sentence into a parsed-structure. We have used a parser developed at Stanford [9] and another written at MIT [10]. The output is a linked-parse which is then analyzed for the types of knowledge that match a particular frame. If a match is found, we create a thing in the form of the respective frame to store that knowledge. There are several methods we have used to do this conversion—each has potentials and shortcomings. Adam Kraft [11] and Michael Klein [3] developed learning programs during their MEng work at MIT. Patrick Winston has written sets of rules to do the conversions in a consistent manner. We have used all of these methods. Regardless, once the thing is created, it is passed on to the memory where my program processes and stores it, as described in this thesis.

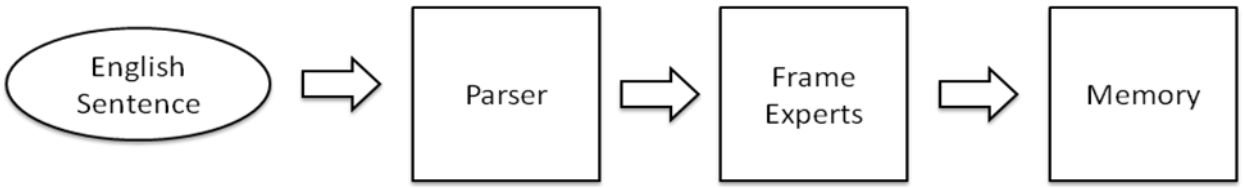


Figure 14: Another look at Genesis

Early Attempts: Self-Organizing Maps

Now that I've described the data that the memory operates on, I will outline the methodology used to develop the current memory system. The earliest goal I had was to generalize from specific experiences to general statements of knowledge. The information is already there in the frames: I simply needed a mechanism for combining similar frames together to extract overlapping knowledge. The data structure that presented itself was the self-organizing map [12].

Self-organizing maps are designed to do two things:

1. Place similar data together in space
2. Modify older data to make it more similar to new data

A classic example of a self-organizing map is that of a two-dimensional image. Each pixel on the plane contains a color. This color is a set of three values: red, green, and blue. We then add new colors to the map. Each new color is placed such that the difference between it and the surrounding colors is minimized. Then, we modify the surrounding colors slightly to make them more like the new color. Over successive inputs, this makes the map organize itself into regions of similar colors.

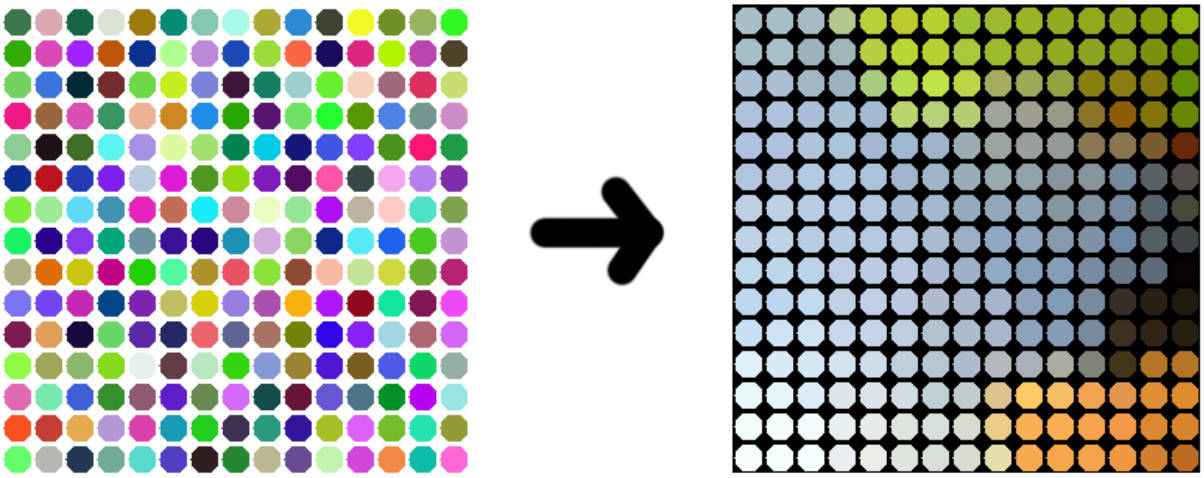


Figure 15: A self-organizing map. The left side is the initialized map; the right side is after the map has had a number of new colors placed in it. Similar colors have clustered together. This image is from Patrick Winston's teaching material [12].

Self-organizing maps seemed like a good choice for a memory system that needs to make generalizations. Each experience has a distance from every other experience. I add an experience to memory so as to be "near" to its neighbors, growing the size of the map. Then, I change the neighboring experiences by modifying their threads to make them more like the new experience. Old experiences of a robin flying to a tree are generalized into a bird flying to a tree. I implemented this system and it worked well in situations with a small number of inputs.

However, there are limitations to self-organizing maps that caused me to eventually reject them. First and foremost, the act of modifying experiences meant I was losing data with time. This has biological plausibility (after all, our memories often decay with age)—but it does not seem like the mechanism for learning generalizations should *require* forgetting old details. It also seems like we humans can recall details we did not believe important at the time if they suddenly fit with new experiences. This means that old memories stay around in one way or another: for my memory, this meant I needed to keep every experience, and modify only the way it was accessed.

The second problem I encountered with self-organizing maps, which was ultimately responsible for me abandoning them altogether, was poor performance. In the example of pixel of colors on a plane there are only two dimensions. In my memory, each experience is the size of the number of things it contains and each thing potentially could have many threads. The dimensionality of experiences, in contrast to colors, is large. The thing-distance metric I described in Part 2 is an expensive computation; as this distance calculation was preformed many times on many experiences, the system simply became too slow both for demonstration needs and from a biologically plausible perspective.

In the end, I scrapped the memory's self-organizing maps in favor of a new way to cluster and compare experiences: chains. This allowed me to develop new and faster distance metrics. The main

ideas from self-organizing maps, namely the self-clustering of data and blurring together of similar data, are extremely important concepts. I used them in my design and implementation of the chains.

Ideas towards a Better System

In the design of my memory system, I incorporated ideas from self-organizing maps, but I also drew inspirations from the ideas and technologies around me. These ideas come from:

1. Michael Klein's Lattice-learning
2. Patrick Winston's Near-miss principle

Lattice-learning provides me with a comprehensive method to classify data given positive and negative examples. By using this algorithm, I am able to abstract away the details of comparing threads and focus on the overall comparison of experiences. This allowed me to develop the idea of a chain. Lattice learning is also significant because it follows the principle of "keep everything". Self-organizing maps broke that principle and they had to go. Lattice-learning is able to weigh old and new data equally—an idea that I carry forward into all of my memory system.

The near-miss principle is important to defining how my memory system would learn. I use it to grow the scope of chains. Lattice-learning alone only changes the specificity of experiences that would match a given chain; near-misses allow me to grow the scope in a way that lattice-learning does not.

To illustrate the different learning styles of lattice-learning and near-miss learning, consider the following example. I have two statements: one, that a robin can fly; two, that a dog cannot. This creates a chain with a positive example thread of a robin, and a negative example thread of a dog.

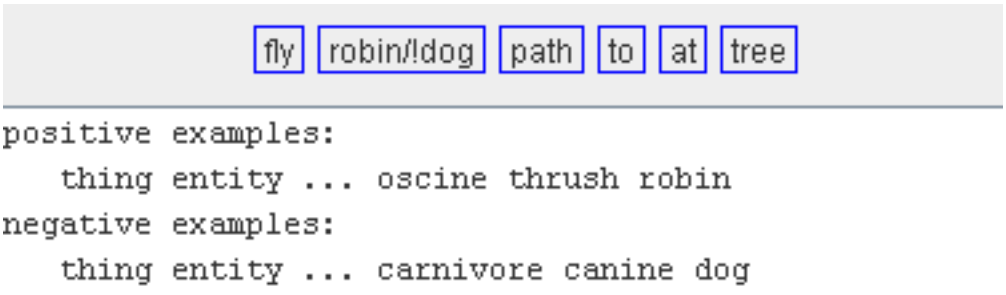


Figure 16: The now familiar chain describing a robin, but not a dog, flying.

Lattice-learning lets us learn whether or not a new thing fits with the current chain—is the new thing supportive of the conclusion presented by the chain or not? For instance, I could have the input stating that a bluebird can fly. There is a strong match to the existing chain: there is no perceived conflict from adding a bluebird thread as a positive example in the second cell of the chain. Lattice-learning thus used the information in the existing chain to "learn" that a new statement is reasonable and consistent.

The near-miss principle allows us to modify the scope of the chain. Say now that I provide the statement that a penguin cannot fly. Lattice learning would consider this to be in conflict with the existing chain—bats are more like dogs than birds, and so lattice-learning would not classify a bat as something that could fly. However, near-miss learning comes into play. The new statement is a perfect match in all other respects to the existing chain: both talk about animals flying. Thus, the bat thread is the only point of contention—we call this a near-miss. By the near-miss principle, I add the statement to the chain anyway. Thus, a single chain now contains information that lattice-learning would have previously found false. Near-miss learning has allowed the chain to grow in a non-intuitive direction.

These two techniques—lattice-learning and near-miss learning—are essential ideas that I sought to encapsulate in my memory system. They are the guiding principles to clustering and learning from similar pieces of experience. Once I had them as tools, my memory's task was to apply them appropriately and significantly to accomplish commonsense reasoning.

Extensions to the Memory

Though I have built a memory system that performs some commonsense reasoning, work is far from complete. The resulting program meets the goals laid out in the introduction as I will discuss in Part 4. However, there is certainly room for increasing the robustness of the memory and for increasing the scope of knowledge it can handle. At this stage, I regard the memory system as a proof of concept but not as a fully functional tool. There are a number of ways the memory could be extended to increase its capabilities.

One obvious but unexplored course of extension to my memory is by improving the filtering of predictions. Predictions are made liberally and there are few sanity checks or methods for ranking quality. For example, say the memory hears that when a mouse appeared, a cat chased it. Given next that a dog appears, the memory might very well predict that the cat will also chase it! The line of reasoning that led the memory to this prediction is via a clear analogy, and we can hardly find fault in it. However, it is reasonable to expect some sort of filtering mechanism to reject illogical or historically false predictions. A very good extension to the functionality of the memory would be to build such a filtering and ranking system.

The second area that calls loudly for further work is question answering. This thesis presents little more than a framework for answering questions. The framework is the idea of strategies, which are wrappers around predefined interactions with items in memory. The few strategies I supply are far from the complete number that a robust system would require. It would be of great use to develop further strategies and improve the mechanism for linking English-language questions to appropriate strategies.

Finally, there is an area outside of the scope of the memory where the memory is already proving quite useful. This is in the task of disambiguation of identical words. For instance, if I say that a hawk appeared in Washington's mall, am I referring to the bird or the militarist? There is no way to discern from the English sentence alone. The trick used to good effect in Genesis is to query the memory and

see whether experiences similar to the one in question talk primarily about a bird or a militarist. If the statement disambiguated to militarist is very similar to items in memory, but the statement disambiguated to bird is not, then we choose that disambiguation. Code to do this is currently written and functioning. Disambiguation is a good demonstration of how my memory system can solve problems in an artificial intelligence system without difficultly. In addition to expanding the work done on making predictions and answering questions, there are many interesting projects that the memory system makes easier to accomplish.

PART 4: Results

I began my thesis work with three goals in mind: to infer generalizations, to make predictions, and to answer questions. In this section I will demonstrate how well the resulting memory system accomplishes them.

Satisfying Goal 1: Generalizations

First, I will discuss the goal of inferring generalizations from individual experiences. In my mind, this is the most fundamental part of what we call common sense. The example of this goal presented in the introduction was:

- *Given that I observe several types of birds flying, I conclude that all birds can fly.*

This is accomplished neatly through the clustering process done by the chains in main memory.

Here is the result of just such a test:

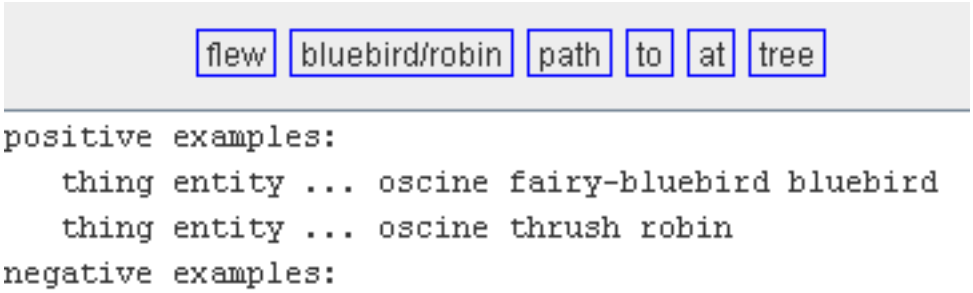


Figure 17: Generalizing from robin and bluebird to bird.

As you can see, two sentences, one about a bluebird and one about a robin, have been clustered together in a single chain. Any statement about any type of bird will now match this chain; thus, Genesis will believe that birds can fly to trees. This generalization has been made from only two examples: a feat we humans can pull off, but many artificial intelligence learning techniques cannot.

Satisfying Goal 2: Predictions

The second goal of the memory system is to make predictions. These predictions are the kind we make as part of our commonsense reasoning. The example originally proposed in the introduction is that:

- *Given that my dog barks at the mailman, I predict that my dog would also bark at a burglar.*

We humans make this sort of simple prediction all the time, and it would be reasonable to expect Genesis to do the same. First, Genesis is given the English sentence:

the dog barked because the mailman appeared.

This results in the chain:

This knowledge is processed and stored in the memory's causal graphs. Later, Genesis is given the sentence:

the burglar appears.

My prediction system queries the memory and notices that a burglar appearing is very similar to a mailman appearing. It uses the causal graphs to make the appropriate analogy, returning a prediction:

As you can see, Genesis has indeed predicted that a dog will bark, exactly as hoped.

Satisfying Goal 3: Answers

Finally, my memory system should be able to answer questions. I have developed a framework to do this and instantiated that process for a few simple examples. In the introduction of this thesis, I proposed answering a question of this form:

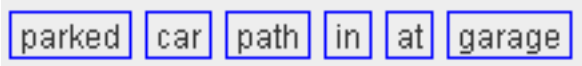
- *Where is my car?*

Given, of course, an experience that describes where my car is. That experience could have happened arbitrarily long ago in the past; it is important that the answer nevertheless come quickly and without difficultly.

First, then, I give Genesis the sentence:

I parked my car in the garage.

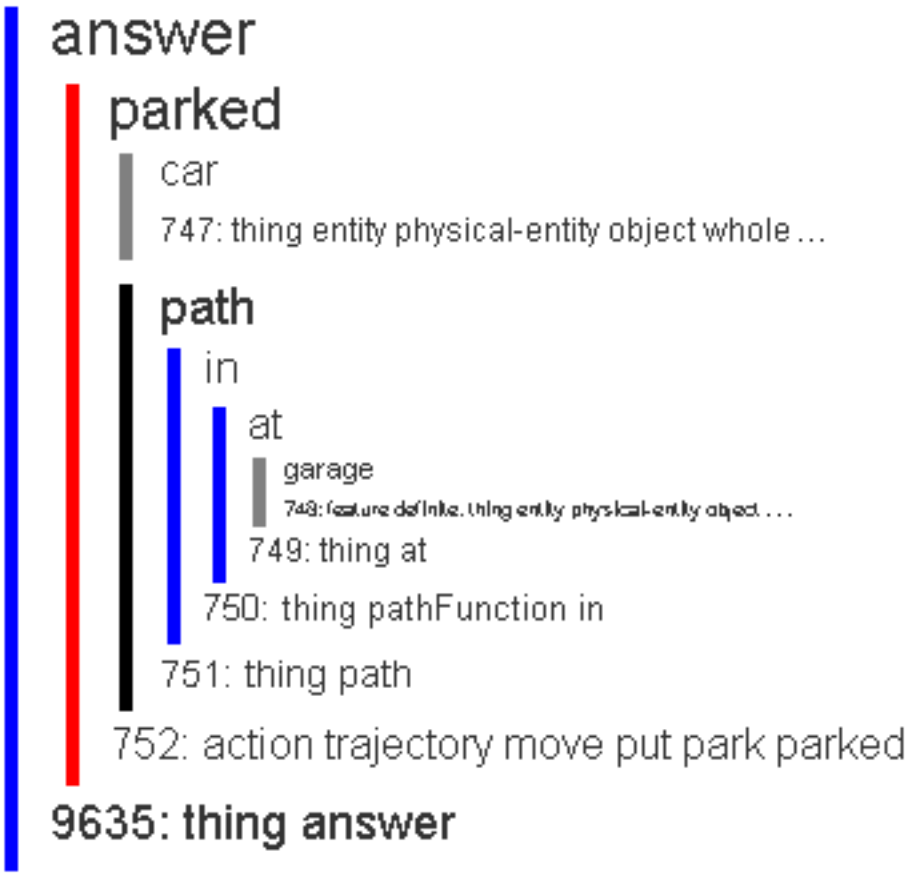
That sentence is parsed and forms the chain:



The chain is stored in memory. I then loaded approximately 100 other sentences in Genesis, all unrelated to cars. Finally, I asked the query (with apologies for the tortured grammar):

Where did the car parked?

Genesis parses this into a question frame and sends it off to the Question Expert. There I use the location strategy to query memory and find the most recent trajectory that discusses the car. That trajectory contains the answer to the question, which Genesis promptly returns:



To the garage.

This is exactly what we would hope to see.

Conclusion

I set out to build a memory system that performs commonsense reasoning. This is important because we humans undertake complex and often convoluted computation with ease. Commonsense reasoning, by definition, is the type of reasoning we find simple, logical, and universal. But it is not, it turns out, easy to implement in a computer system. Our minds do a considerable amount of work behind closed doors; we are conscious only of the results of these processes. By simulating such mental activities in an artificial intelligence system, I hope to further our understanding of our own human intelligence.

Contributions

I divided commonsense reasoning into three parts: inferring generalizations, making predictions, and answering simple questions. My contribution to our understanding of human intelligence is by providing computational mechanisms to accomplish these three goals.

- I have introduced *chains*, which cluster together similar experiences. This allows for generalizing collections of experiences into blanket statements of knowledge.
- I have coupled the concept of chains with a set of graphs to store historical and causal knowledge. This allows the memory system to make predictions by means of both historical precedence and simple analogies.
- I have designed a framework for answering commonsense questions by developing a set of strategies for interacting with the memory system. These strategies are matched to questions to provide answers.

It is my hope that these contributions can be built upon by successive artificial intelligence researchers both within the scope of the Genesis project and in the wider community. I believe that by

building models that shed light on human intelligence we can improve our understanding of ourselves and our interactions with each other in societies.

References

[1] L. Vaina and R. Greenblatt, “The Use of Thread Memory in Amnesic Aphasia and Concept Learning.” MIT AI Lab: 1979.

[2] R. Jackendoff, *Semantics and Cognition*. MIT Press: Cambridge, 1983.

[3] M. T. Klein, “Understanding English with Lattice-Learning.” Master’s Thesis, MIT, 2008.

[4] P. Winston, “Learning Structural Descriptions from Examples.” PhD thesis, MIT, 1970.

[5] S. Needleman and C. D. Wunsch, “A general method applicable to the search for similarities in the amino acid sequence of two proteins,” *Journal of Molecular Biology,* vol. 48, iss. 3, pp. 443-453, March 1970.

[6] P.Winston, *6.034 Demonstrate*. [Software]. MIT CSAIL: 2009.

[7] G. Borchardt, “Causal Reconstruction.” MIT AI Lab: 1993.

[8] G. A. Miller, WordNet 3.0. Princeton University, 2009. [Software]. http://wordnet.princeton.edu/.

[9] *The Stanford Parser: A Statistical Parser.* Stanford NLP Group: 2009. [Software].

[10] B. Katz, *START*. MIT CSAIL: 2009. [Software]. http://start.csail.mit.edu/.

[11] A. Kraft, “Learning, Using Examples, to Translate Phrases and Sentences to Meanings.” Master’s Thesis, MIT, 2007.

[12] T. Kohonen, *Self-Organizing Maps.* 3rd ed. Springer Series: New York, 2001.