ABSTRACT

Title of Thesis: Towards Relational Theory Formation from Undifferentiated Sensor Data

Marc Pickett, Ph.D., 2011

Thesis directed by: James T. Oates, Associate Professor
Department of Computer Science
and Electrical Engineering

Human adults have rich theories in their heads of how the world works. These theories include objects and relations for both concrete and abstract concepts. Everything we know either must be innate or learned through experience. Yet it's unclear how much of this model needs to be innate for a computer. The core question this dissertation addresses is how a computer can develop rich relational theories using only its raw sensor data. We address this by outlining a "bridge" between raw sensors and a rich relational theory. We have implemented parts of this bridge, with other parts as feasibility studies, while others remain conceptual.

At the core of this bridge is Ontol, a system that constructs a conceptual structure or "ontology" from feature-set data. Ontol is inspired by cortical models that have been shown to be able to express invariant concepts, such as images independent of any translation or rotation. As a demonstration of the utility of the ontologies created by Ontol, we present a novel semi-supervised learning algorithm that learns from only a handful of positive examples. Like humans, this algorithm doesn't require negative examples. Instead, this algorithm uses the ontologies created by Ontol from unlabeled data, and searches for a Bayes-optimal theory given this "background knowledge".

The rest of the dissertation shows in principle how Ontol can be used as the "workhorse" for a system that finds analogies, discovers useful mappings, and might ultimately create theories, such as a "gisty" theory of "number".

Towards Relational Theory Formation from Undifferentiated Sensor Data

by

Marc Pickett

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Ph.D.
2011

For Mom and Dad

ACKNOWLEDGMENTS

I'd like to thank the following people:

Tim Oates, for superhuman patience as I kept worrying about whether I was working on "important problems".

Matt Schmill, for helping to keep my research goals reasonable. Otherwise, I still might be formulating my problem setup.

The rest of my committee: **Rob Goldstone**, **Tim Finin**, and **Sergei Nirenburg** for discussion and reading drafts of my dissertation.

James MacGlashan, with whom I discussed so many ideas, I'm no longer sure which are mine and which are his, so the MacGlashan Transform is named after him. 90% of this dissertation was developed on James's whiteboard.

Bill Krueger, **Tom Armstrong**, and **Max Morawski** for acting as sounding boards for my ideas.

Marie desJardins, who got me interested in traffic jams during her "Emergence" seminar.

Niels Kasch, to whom I owe Chapter 2, as this chapter is essentially a transcript of one of our whiteboard discussions.

Patti Ordonez, who showed me the power of deadlines.

Gabe Chaddock, about whom I could write a book.

The rest of my labmates during my time at UMBC: **Blazej Bulka**, **Don Miner**, **Sourav Mukherjee**, **Josh Jones**, **Soumi Ray**, **Niyati Chhaya**, **Mitesh Vasa**, **Yasaman Haghpanah**, **Joe Catalano**, and **Shamit Patel**.

For everything: **Mom**, **Dad**, **Jason**, **Matt**, and **Amy**.

And finally, I'd like to thank **Gina** for endless moral support, putting up with me coming home as a zombie for months, teaching me how not to reinvent the wheel, and teaching me how to merge. It certainly wouldn't have been so easy without you. Thanks Shammy, ILU!

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

PROBLEM STATEMENT, RELATED WORK, AND CONTRIBUTIONS

> From the child of five to myself is but a step. But from the new-born baby to the child of five is an appalling distance.
>
> –Leo Tolstoy (1828-1910)

Much of how we describe our world incorporates relations among abstract concepts. For example, we understand news stories about the "trade balance" between "China" and "The United States", we understand the meaning of the sentence "Fear is the destroyer of dreams.", and we understand that "NP-complete problems don't necessarily require an exponential amount of memory to solve". Somehow, we have theories of the world that are rich enough to understand all these concepts. Likewise, for a computer to be considered intelligent, it also must have theories that are rich enough to represent and understand such concepts. Anything that a computer knows must either be hand-coded by a human or must be learned through its experience in the world. Efforts have been made at hand-coding this kind of knowledge with

limited success (Witbrock *et al.* 2005), (Lenat & Guha 1990), but it seems clear that anything approaching human-like requires a high degree of learning, as it's unlikely that anything about China or NP-completeness is encoded in our DNA.

Traditional approaches to Artificial Intelligence focus on selecting an application and then constructing representations for that domain by hand. These approaches are problematic in that they require much labor intensive knowledge engineering. Furthermore, these systems tend to be brittle, often failing when they encounter unanticipated situations. A newer approach to AI is to have a computer develop its representations autonomously. In this alternate approach the computer is provided a minimal amount of knowledge (implicit or otherwise) about the world and is expected to learn and develop a conceptual structure from large amounts of raw sensor data over a long period of time. In this approach, the computer is viewed as a "robot baby" (Cohen *et al.* 2002) because, like human infants, it comes into the world knowing very little and must learn much of what it knows from its environment. The robot baby approach is attractive because it requires little knowledge engineering and is robust because the computer learns to adapt to unanticipated situations. This approach also directly addresses the Symbol Grounding Problem (Harnad 1990) —the problem of creating meaning using only a set of meaningless symbols— by directly grounding all a computer's knowledge in sensory data.

The "knowledge-free" property of the robot baby approach has several advantages. It keeps the system general, and it makes it easy for us to feed data *about* the system back into the system (for metacognition). Furthermore, with the decreasing price and increasing ubiquity of sensors, there are modalities where domain-knowledge is non-existent or difficult to get, such as the traffic data in Figure 1.1. All these sen-

sors provide potentially useful data that go unused for lack of knowledge of how to use it. Keeping our algorithm knowledge-free also simplifies the model in that we have only a handful of representational mechanisms, instead of a different mechanism for each domain. We hypothesize that an algorithm that can go from raw sensors to a perceptual system can also go from a perceptual system to higher-level concepts.

Under our approach a computer is explicitly told very little about the world, but, if it's to be intelligent, it somehow must be able to understand abstract concepts like the examples mentioned above. Thus, we have the central question that this dissertation seeks to answer: *How can a computer develop rich relational theories using only its raw sensor data?*

The robot baby approach faces several challenges. For any *specific* domain, a human-engineered approach usually outperforms an autonomously developed approach. But we're interested in a general intelligence. It's doubtful that a system engineered to play chess will also do well at a voice recognition task. If we're ever going to achieve human-level machine intelligence, we need to have a general-purpose learning machine. Beyond a certain level of complexity, any autonomously-generated representation will be difficult for humans to interpret. For example, the weights learned by feed-forward neural networks are often difficult to interpret (Garson 1991). For this reason, it's difficult to directly provide domain-knowledge from human experts. We must remember that this is a limitation suffered by *human* babies also. This property may be inherent for any sufficiently complex representation in general, whether autonomously acquired or knowledge-engineered. Therefore, we propose to make our representation-acquisition algorithms as simple as possible, so we can more easily be sure of their validity and the validity of the representations they develop.

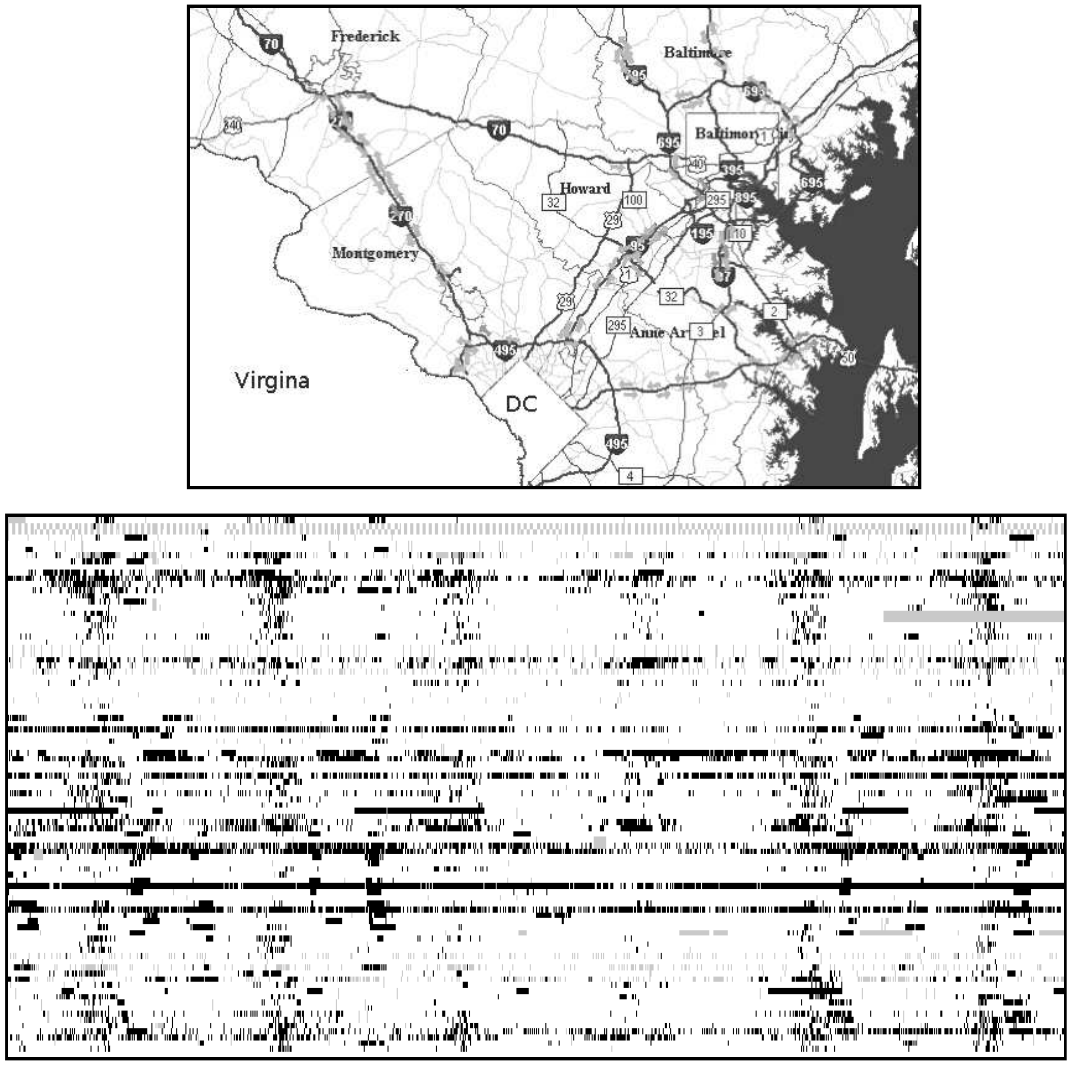


FIG. 1.1. **Traffic speed sensor data.** The top image is a map of automobile traffic speed sensor locations. Below, is a time series of the traffic values taken every 5 minutes for a week. The y axis are the 87 sensors in an arbitrary order. The x axis is time. Black areas are where the average traffic speed over the 5 minute window is less than 25 miles per hour. Grey areas are missing data. We can see the morning and evening rush hours as dark evenly spaced bands.

This is an argument *for* a knowledge-free approach. This keeps the system simple. A general rule-of-thumb for designing learning systems may be summarized as "an ounce of (human-provided) heuristic is worth a pound of search". If we allow knowledge-engineering at any level, we won't be sure if the success of the algorithm is due to the knowledge-engineering or due to the system's own representation-acquisition. Thus, a core challenge we face is resisting the temptation to engineer a domain-specific system, which we address by forcing our algorithm to work on a wide range of domains, especially domains about which we have little domain knowledge. Another disadvantage of the robot baby approach is the large amount of data needed for the system to build its own representation. However, this is becoming less of a problem as sensors are becoming cheaper and the sheer amount of data being generated by these sensors increases. Furthermore, data needed by the robot baby doesn't need to be labeled and requires a minimal amount of massaging, making it inexpensive to acquire. The robot baby approach also requires a learning period —possibly years— before it becomes an expert about a domain. This also may be an inherent property of learning about complex domains. There's a trade-off between plasticity and innateness. Humans are able to learn to play abstract games, like Chess or Go, which didn't exist for most of our evolution. But human babies also require several years of experience to get to this point. In contrast, antelope are born knowing how to run, but it's doubtful an antelope will ever become a Chess expert, no matter how many years of Chess experience they're given.

In building a robot baby, we must address The Squiggly Lines Problem: The robot doesn't know what its sensors mean, it just has the sensor readings over time, which, when plotted, look like a bunch of squiggly lines, as shown in Figure 1.2. For example, consider a Pioneer robot with sonar sensors. Suppose we're given only plots

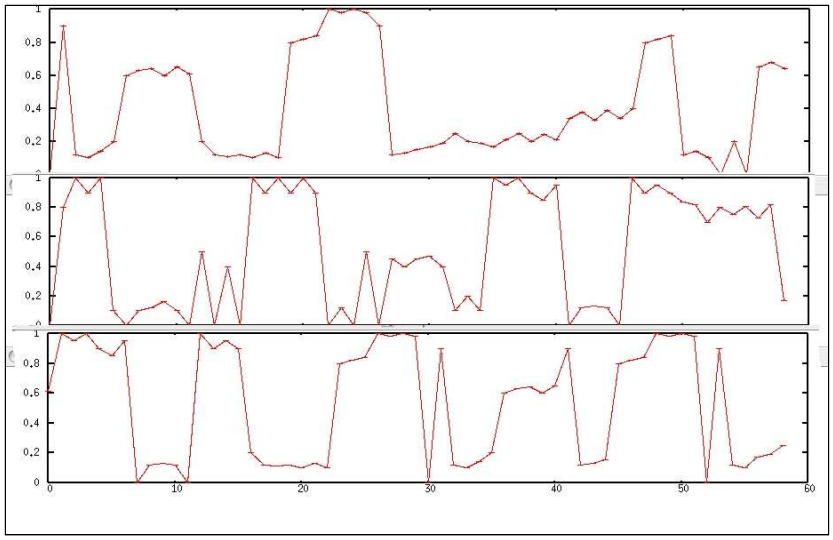


FIG. 1.2. **Squiggly Lines.** This shows data over time for 3 sensors.

of these sensors' readings, but no other information. We're not told which sensor is which. We don't even know that these are sonar sensors from a mobile robot. As far as we would know, these could even be readings from (a simulation of) a mobile robot in a 5 Dimensional world. If the Pioneer moves past a trash can, the base sonar readings would be qualitatively different depending on whether the trash can is upright or knocked on its side. The Squiggly Lines Problem is going from these squiggly lines to a theory of the world that, if applied to the Pioneer's data, would include 3D objects (and maybe a simple theory of 3D physics).

The theories that people hold about the world are qualitatively different from those held by traditional knowledge-engineered systems. For example, people have "gists" for integers higher than about 10. We know that 56 is close to 57, but for most of us a pile of 57 marbles is nearly indistinguishable from a pile of 56 marbles. We have a loose mapping between a pint-sized jar of marbles and the *number* representing

the quantity of marbles in it: we know that it's more that 20 marbles, but less than a billion. We know that 50 degrees Fahrenheit is too chilly for jeans and a t-shirt, but too warm to wear an arctic snow-suit. When we simulate 56 marbles rolling across the floor in our minds, we might simulate a single entity called "a bunch (not even a definite number) of marbles". Our "gisty" or intuitive theory of numbers is different from that of number theory or a computer's representation of numbers. A question this dissertation helps to answer is: How can a computer develop gisty theories, such as a gisty theory of integers?

The initial idea for this work was sparked by a "morphable face model" that recognizes human faces by using a 3D model of them (Blanz 2006). The computer was provided with a basic model of rotation, lighting, and 3D shapes. The morphable face model was an engineering endeavor, so there was no shame in hand-holding, and this system had plenty of it. This gave rise to the following question: How could a computer ever develop *on its own* a "theory" of 3D faces, like that used in the morphable face model, given only 2D images? That is, in principle, how can we build a *bridge* spanning the vast gap from raw sensor data to rich domain theories, such as the basic 3D physics model in (Blanz 2006), *without* knowledge engineering? We suspect that the mechanisms introduced in this dissertation are sufficient for this.

In this dissertation, we provide an outline to the answer for this question, with various parts implemented and tested. We provide the basic design for this bridge, and have constructed various segments. A full implementation of this bridge may take decades of work, so some parts of our bridge are steel and concrete, some places are wooden sticks, and some places are only conceptual. The point of this dissertation is to provide a framework, with implementations to show the feasibility of this approach.

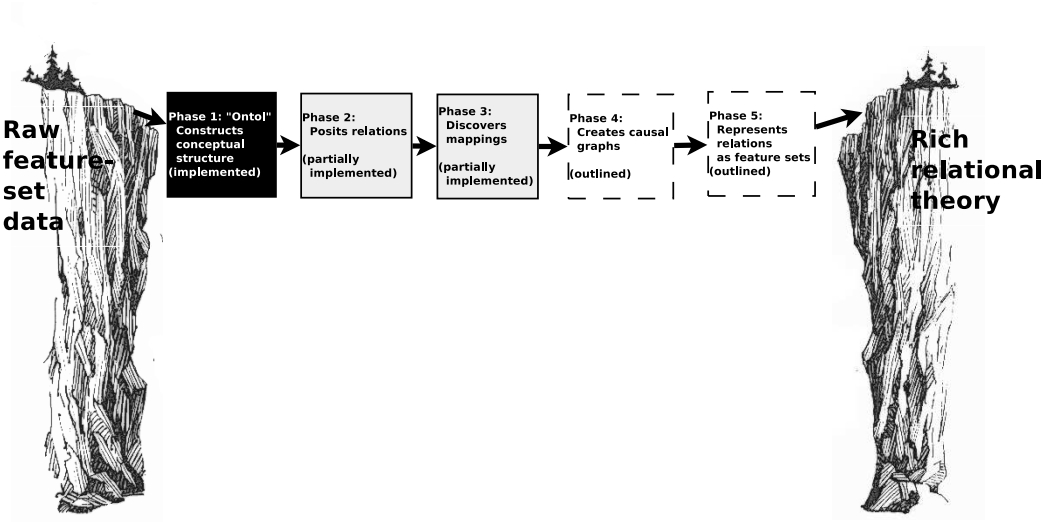


FIG. 1.3. **"The bridge".** The core parts of Phase 1 are implemented. Parts of Phases 2 and 3 are implemented, while Phases 4 and 5 are conceptual.

Our design is shown in Figure 1.3. The bridge consists of 5 phases. Phase 1, the major technical contribution of this dissertation, is for our system to learn a conceptual structure, called an "ontology". We call this ontology-builder "Ontol". The remaining phases are only partially implemented, as a full implementation is beyond the scope of this dissertation. Phase 2 introduces relational concepts by finding parts of the ontology that are analogous, then inducing parameterized schemas from the overlap of these parts. Phase 3 discovers mappings using the concepts discovered from Phase 2. Phase 4 finds overlap among systems of mappings to create parameters for mappings. Phase 5 represents relational structures in a manner such that these can be fed back into Ontol, effectively doing a version of metacognition.

1.1 Assumptions

This dissertation is based on the following assumptions:

I. *Hand coding representations is a doomed approach to AI* (or at least it's not the best way to go about AI).

The traditional approach to AI is where a person will pick an application, then construct representations for that domain. The problem with this is that it takes a good deal of knowledge engineering (which is labor intensive), and the system will fail if something happens that the person didn't anticipate.

An alternate approach is to have the computer develop its representations autonomously, and this is what we're proposing to do in this dissertation. Therefore, a driving force in developing our algorithms is domain independence. To guard against tailoring our algorithms to any single domain, we have a suite of disparate domains.

The breadth of these domains is used to illustrate the generality and knowledge-free property of the algorithms. Each domain is represented in feature-set form, but beyond that, we make no modification of either the domain or our algorithms.

II. *We're not allowing the learner to take actions to affect the world.* That is, our system will be a passive observer. Our system has neither actions, goals, or temporal information. This is for simplicity because we believe it's possible to learn models without these. Our system should be able to leverage these if they're available, but the simplest case would be the system just as an observer.

Our view is that allowing an agent to take actions confounds the problem of

making a robot baby. People can learn a good deal just by observation. For example, the fields of geophysics and astrophysics are largely observational. Just because we can't perform experiments on stars doesn't mean that we can't build a theory about how they work, even a *causal* theory.

If we allow the agent to take actions, then we would have to address how the agent plans and performs experiments, and for now we'd like to focus on the learner/reasoner under the assumption that there's a clean way to add in actions later. Once we have a working theory-maker, then we assume that we can just add the actions as extra signals, and the system will build a theory of how actions affect the world. It can then use this theory to take meaningful actions.

Although statisticians have shown that you can't be 100% sure of causality without causal experiments, Pearl (Pearl 2000) points out that one can be *pretty sure* about a causal theory given non-causal data. Essentially, the most parsimonious causal theory that explains the data is probably the correct one. There is also evidence that 4-month-olds have causal concepts that 3-month-olds don't have, despite their limited abilities to affect the world (Leslie & Keeble 1987), so the necessity of actions to develop a world model (for example in Drescher's model (Drescher 1991)) is questionable.

Also, if our system can't take actions, and the data is unlabeled, we're left with an unsupervised learning problem, and the question arises as to what will be used to drive learning. This leads to the next Assumption.

III. *A shorter theory is a better theory.* This assumption gives us a goal to go for: the shortest theory that explains the data. This is the driving force in our algorithm, which essentially does compression.

We later question this assumption in Subsection 8.2.1. Essentially, there's a

trade-off of time and memory. For example, given a few inference rules and Euclid's 5 postulates, we can derive all of Euclidean geometry. However, in practice, we'll probably want to *cache* some of the lemmas etc.. We could combine time and memory into a single measure for how good a theory is (e.g. "50 time units equals 1 memory unit"), but there may be a more elegant way to address this issue.

IV. *Optimality is unnecessary for intelligence.* People make mistakes, and our computers should be allowed to too. Many of the problems we're working on are NP-hard or worse, but we mustn't fall victim to what Seymour Papert calls the "Superhuman Human Fallacy", which is the belief that *people* can perfectly solve NP-hard problems quickly.

    Ultimately, we want our runtimes to be $O(n \log n)$ or better where n is the size of the data. $O(mn \log(n))$ algorithms should be allowed if these algorithms are parallelizable among m "processors". Here we consider each region of our ontology (which we refer to as a cortical region, following (Hawkins & Blakeslee 2004)) as its own processor, where "region" and "processor" are both loosely defined.

V. *For domains we're interested in, we have access to a large amount of data.* We want our computer's knowledge to come from ample domain data, not from human experts. Over the first few years of life, humans are exposed to a tremendous amount of sensory inputs. Likewise, we assume that large amounts of data are cheap to acquire for our computer. Since our algorithm is knowledge-free, we don't have to label anything. Furthermore, our algorithms are based on a prediction framework that allows us to "predict" missing data.

VI. *It's possible for a system to develop a full model of the world even if it starts with very little innate knowledge.* It's clear that animals and possibly humans are born with a good deal of innate knowledge. For example, antelopes are born knowing how to run and avoid predators. Here the innate knowledge has been provided by millions of years of evolution. But we're trying to build smart robots, and we don't have millions of years of evolution to put in them innately. The innate knowledge is helpful and *evolutionarily* necessary, as it reduces the amount of time to bootstrap learning, but it may not be necessary for intelligence.

Thus, we seek to find the minimal amount of innate knowledge necessary for bootstrapping intelligence. This might be quite small. Using Assumption III we can conceptually solve the Squiggly Lines Problem (but with an exponential time solution) in a manner similar to that of Hutter's Universal Artificial Intelligence:

(a) Give the robot a representation framework that's expressive enough to encode physical objects, causality, 3D physics, etc..

(b) Give the robot *a lot* of data from its sensors, say a few years' worth of sensor readings.

(c) Generate all the theories that are of a complexity less than that of the raw data, and choose the smallest that can be "flattened" to produce the raw data.

We hypothesize that (given enough data, and an expressive enough representation framework) this will find a causal theory that includes naive physics.

Ultimately, the goal of the robot baby approach to Artificial Intelligence is to

bring the computer from the level of a newborn to that of a 3-year-old human, which means the computer would have a "gisty" theory of physics, natural language understanding, and other basic concepts. Once the computer acquires language understanding, it can use this to tap into humanity's vast store of cultural knowledge (both tacit and explicit). We hypothesize that if we develop a theory that's sufficient to bring a robot baby from the intelligence level a of newborn to that of a 3-year-old human, then this theory will also be sufficient for taking the robot from the level of 3-year-old human to that of an adult human.

1.2 Evaluation Criteria

A complete, implemented answer to our question —"How can a computer develop rich relational theories using only its raw sensor data?"— is beyond the scope of a single dissertation, but it's our aim to make progress in answering this question. Therefore, we present the following list of desiderata to help evaluate prospective answers to this question. We split these desiderata into those for our theory of learning, and those for the theory that our system learns from its sensor data. That is, we want to know how good an ontology or theory is.

The goal of this work is not to build the entire bridge (from raw sensor data all the way to a relational theory), but to give a plausible story and to implement and test certain pieces. Both the story and the implemented pieces need criteria by which to be evaluated.

1.2.1 Desiderata for our theory of learning

Here we list desired properties for a theory describing an intelligent system. Earlier works discussing desiderata of intelligent agents ((Sun 2004), (Rosenbloom, Laird, & Newell 1993), (Newell & Simon 1972)) tend to focus on planning and problem solving using a human-provided domain model, and containing autonomous learning and development of representations as an afterthought, if at all. We claim that a stronger emphasis should be placed on ground-level learning.

Since we provide a minimal amount of domain knowledge, **domain independence and generality should be among the top criteria** for a learning and reasoning computer. Therefore, an empirical demonstration of such a computer should contain several disparate (though data-rich) domains with a minimal amount of human-provided data "massaging". A set of domains might contain robot sonar sensor data, a large corpus of text, a series of images, and a simulation of Conway's Game of Life. For each of these, a computer should, at a minimum, autonomously develop an ontology that's useful for characterizing that domain. For example, when given sonar data, the computer may build a hierarchy of motifs. When given images, the computer should develop edge filters, and when given Conway's Game of Life, the computer should develop the concept of a "glider". Ultimately, we'd like to have a computer that will build the theory appropriate for whatever data it's presented with. Given ample image data, the computer should eventually discover physical objects and concepts such as translation, rotation, and scaling. Beyond that, a computer should eventually develop the concept of "degree of rotation", a "gisty" theory of numbers or an intuitive theory of quantity, and it should understand that the degree of rotation is similar to the number of apples in a group in that both are a matter

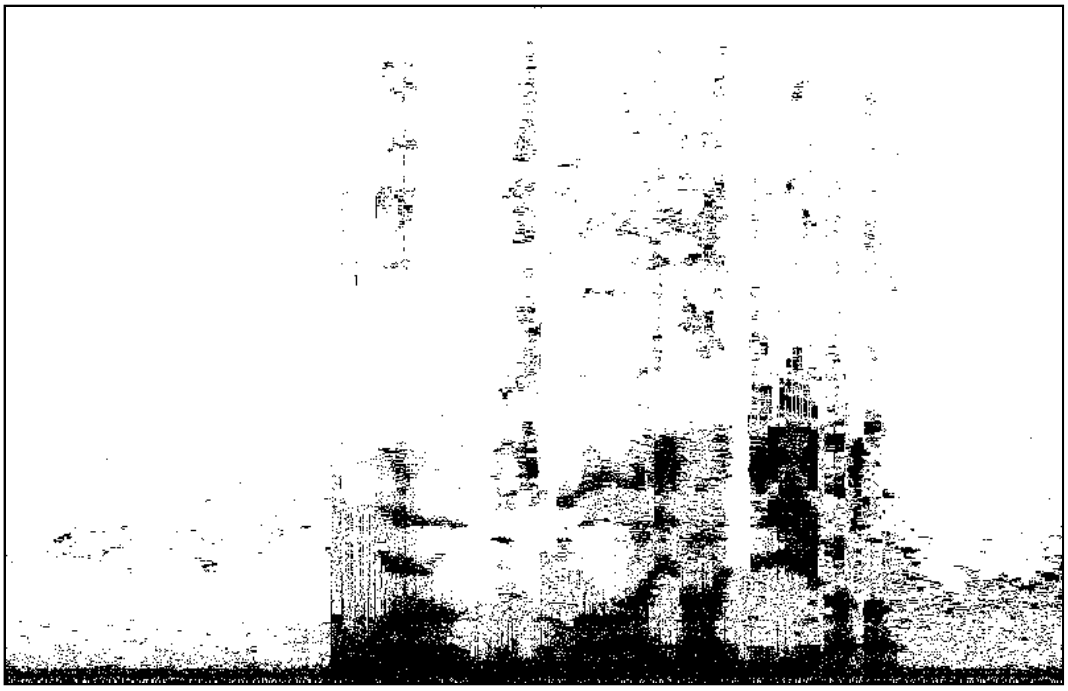


FIG. 1.4. **A spectrogram of the spoken sentence "The world is my idea.".** The x axis is time, and the y axis is frequency.

of degree. The computer should be general enough such that the same computer can develop the relevant concepts for other types of data. For example, if the computer is given enough English speech data, like in the example in Figure 1.4, the computer should discover phonemes, English words, and transformations such as pitch changes, speed shifts, loudness, and different accents, such as "British" and "American" accents. Similarly, we'd like the same computer to discover Chinese words and Chinese phonemes when given ample spoken Chinese data. Such feats are beyond the scope of a dissertation, but we would like to at least have a conceptual story for how they might be accomplished.

Our story should be as simple as possible. This is useful not only because

we want our theory to be scientifically valid according to the principle of Ockham's razor, but also because we'd eventually like to implement the entire system. If a premium is placed on model elegance, building the system will be made easier. In addition, in a real-world situation, such a system will be provided with gigabytes of data and will create a theory with millions of concepts. This amount of data will make such a system particularly tricky to understand and debug, which also places a premium on keeping the system as simple as possible.

In keeping with simplicity, we're looking for a general algorithm with **a handful of mechanisms** as opposed to a large collection of specialized subsystems. In particular, it will be useful to have mechanisms **that work "all the way up"**. That is, the same mechanism that builds low-level concepts from raw sensor data should be able to build mid-level concepts from low-level concepts. This works well with our goal of making the story domain-general: if our system generates low-level concepts from generic sensor data, then we can treat the low-level concepts as sensor data from which to create higher level concepts and so on.

Since we ultimately want to implement and run a working system, **our story needs to be implementable**, and it needs to **require reasonable computational resources**. A story that required super-polynomial time or memory would therefore be unacceptable.

We're interested in how a *computer* can develop a relational theory from sensor data. It's conceivable that the story might be quite different from how people learn. That being said, if our theory is **biologically or psychologically plausible**, such a story might lend insight on human intelligence. Furthermore, since our only existing example of intelligence is biological, it would add strength to our story if it

corresponded to psychological or neurological models of representation and learning, though this correspondence isn't a requirement.

The desiderata for *our* theory of learning are summarized here:

1. The theory should be as simple as possible.
2. The theory should focus on having as little innate knowledge as possible.
3. The system needs to be implementable and require reasonable computational resources.
4. The theory should be testable across a wide range of domains with no changes to the algorithms.
5. Though not necessary, the theory will be strengthened by biological and psychological plausibility.
6. The theory needs to offer a plausible story for how the gap is bridged from raw sensor data all the way to a relational theory of the world.

1.2.2 Desiderata for the theory learned by the system

To answer the question more precisely of exactly *what* an intelligent computer should do with its data is perhaps tantamount to answering the question of what intelligence is. It has been suggested that a core purpose of intelligence is to concisely characterize a set of data (Wolff 2003), (Hutter 2004). That is, given data, an intelligent computer should generate a model that best compresses the data. This is the principle of Minimum Description Length (MDL). It is fundamentally equivalent

to Ockham's Razor, which says, in effect, that "The shortest model (that predicts the data) is the best model.". If we assume that the prior probability of a model is inversely proportional to the exponent of its description length, then Ockham's Razor is also fundamentally equivalent to the Bayesian principle that states that "The most probable model is the best model.". In Subsection 8.2.1, we question the use of this metric, but it's straightforward to evaluate, and is a good rule-of-thumb for creating concepts that correspond to what people might discover.

We hypothesize that many concepts that people have are objectively useful because they allow us to characterize the world. For example, the class of mammal isn't just a human invention, but is objectively useful for characterizing animals. If an alien visitor observed our planet, we hypothesize that given enough experience, the alien would independently distinguish between mammals, fish, and reptiles. To be sure, there are plenty of concepts that are task dependent. For example, an army general might look at a mountain range as a defensive barrier while a mountain climber might look at the same range as a series of peaks to be climbed. But we hypothesize that there are a core set of concepts that are useful for a wide variety of tasks. Intuitively, this makes sense because a human needs only a handful of examples to learn a concept such as what an army tank is, whereas most supervised learning techniques require hundreds if not thousands of examples. Even if shown hundreds of photographs, labeled "with" and "without" tanks, a computer is as likely to conclude that the concept of "tank" corresponds to a cloudy day as it is to come up with the actual concept of "tank" if the "with" pictures were taken on a cloudy day and the "without" were taken on a sunny day. A human probably wouldn't make this mistake. We suspect that humans do this by leveraging their previous knowledge that's been gained through their experience in the world. Among other things, this dissertation

demonstrates this principle by showing that an ontology built through unsupervised learning can be used to greatly improve performance on supervised learning tasks. That is, we can use our ontology to learn concepts from a tiny number of labeled exemplars. Note that we're not getting anything for free: we still have to train the computer using a large number of *unlabeled* exemplars.

For simple domains and test cases, we can simply look at the theory developed and see if it corresponds to human-provided concepts. For example, on a simple animal dataset, we can look at the developed ontology and see if the concepts correspond to mammals, reptiles, etc.. Simply looking at the ontology becomes more difficult for more complex domains because the ontology won't be so easily interpretable if it's formed on vision data, for example. For such cases, we can still evaluate the learned concepts by seeing if these concepts allow us to improve performance on supervised learning tasks, as mentioned above.

But there are some datasets for which acquiring human-provided knowledge becomes difficult. For example, datasets with which few people have firsthand experience and intuitions about, such as the time series of automobile traffic highway speed sensor data shown in Figure 1.1. Our hypothesis is that if, when run on domains about which humans have knowledge, the concepts developed by our system are sensible, then the concepts developed by our system on other domains will also be sensible.

Furthermore, in accordance with (Hawkins & Blakeslee 2004), our system should be able to use its theory to make accurate predictions about missing data, for example.

Since our system is only an observer, we don't have actions or reward, so we can't

use reward as an evaluation metric. But as for supervised learning, the developed theory should be useful for goal-driven tasks, such as reinforcement learning.

In summary, we have the following desiderata for the theory learned by the computer:

1. The theory should allow for compression of the data.
2. Generally, the theory should create concepts that correspond to human-developed concepts. This includes being able to leverage the theory to improve performance on supervised learning tasks with human-provided labels.
3. Our system should be able to use the theory it develops to make accurate predictions about missing data.

This is an overview of evaluation. In general, we'll use compression as the evaluation metric for each part, but will mention how the specifics are spelled out for in each chapter for that part.

1.3 Related Work

Below is an overview of related work for the whole story for going from raw sensor data to a rich relational model of that data. Detailed related work for each chapter will be separate.

There has been relatively little work on creating relational concepts from feature-set data. The work of (Pierce & Kuipers 1997) and (Kuipers 2008) begins with raw

undifferentiated sensors from a mobile robot, and builds a "rickety bridge" from these sensor readings to a map of the physical world. This work presents a sequence of methods for building each step of this bridge: from raw sensors to the physical structure of the sensors, to building a model of the robot's effectors. Although this work shows promise as a demonstration of the principle of knowledge-low learning, the methods implicitly provide plenty of domain specific knowledge to the learner along the way. For example, that the sensors are located in a three dimensional space. Other sensors, such as sound frequency sensors in a microphone, might not have such a physical interpretation. It's questionable whether the same methods would work on vastly different domains without changes to the algorithm. It's also unclear whether these learning methods can be "stacked" to learn more abstract concepts than just those in the physical domain.

The work of (Kemp & Tenenbaum 2008) is more general. As its input, it takes a matrix that's usually a relation among entities, but may also be a set of feature vectors. For this latter case, the algorithm uses entity graphs, which effectively posits a relation among 2 feature-sets if they have many features in common. This approach only posits relations among feature-sets, not the features themselves. So it's unclear that this approach could ever posit a theory of rotation given only a set of visual bitmaps. Given this generative model, the algorithm calculates the probability of the feature-set given the structural model, and uses this to search for the most likely model. The algorithm uses graph grammars to set up a space of structural forms that is then searched over to find the most probable form given the data. Using graph grammars to represent the learnable space of structural forms is a shortcoming because graph grammars can't represent such basic structures as the generalized concept of a clique or a lattice of unlimited degree (Ehrig *et al.* 1999).

There has also been work on a inducing the structure of Bayesian networks from feature-set data (Friedman & Koller 2003) and on inferring the structure of Probabilistic Relational Models (PRMs) (Getoor *et al.* 2002), (Getoor *et al.* 2001b), (Getoor *et al.* 2001a). However, the structures learned by these approaches are relations among features. These approaches don't makes analogies, and they fail to create *parameterized* relations. That is, they propose link structure between features, but, in an example where the relation between (e.g.) `feature1` and `feature2` is the same *type* of relation between `feature3` and `feature4`, they would fail to exploit this property.

In our "bridge" story, we propose that "behavioral isomorphisms" play a key role in creating relational concepts. The work most relevant for this is the work on finding and using analogies. This work has been extensive: for example (Hummel & Holyoak 2005), (Hummel & Holyoak 2006), (Falkenhainer, Forbus, & Gentner 1989), (Gentner 1983), (O'Donoghue 2005), (Dietrich 2000), (Marshall & Hofstadter 1996), and (Hofstadter 1984). There has also been work on using analogy or graph-isomorphism for creating relational concepts, (Gonzalez, Holder, & Cook 2002), (Holder, Cook, & Djoko 1994), and (Nevill-Manning & Witten 1997). However, all of these systems assume our data is in a relational form to begin with. The question of where this relational form comes from is left open. We know of no other work on this problem save what we've already mentioned. Therefore, we suggest that this question needs addressing, for which this dissertation is a beginning, though we make no claim about having fully answered this question.

1.4 Contributions

The main conceptual contribution of this dissertation is providing a plausible story of how we can go from domain-general feature-set data to a rich relational theory of that data. For example, given visual data, we describe how a system might learn to build a theory of rotation or a "gisty" theory of integers. This system is knowledge free (i.e., it works on raw sensor data across a broad range of modalities), unsupervised, and not dependent on time or any other explicit relational information. Furthermore, the bulk of this system is biologically and psychologically plausible as it loosely corresponds to neurological models of the neocortex, such as those describe by (Riesenhuber & Poggio 1999) and (Hawkins & Blakeslee 2004). This provides the skeleton for a bridge from sensors to a rich relational theory. This story is only partially implemented, as a full implementation of this story is beyond the scope of a dissertation. We argue that the mechanisms described are sufficient for constructing even higher-level theories such as a theory of international economics (though efficiency issues would have to be addressed). This dissertation provides this story as well as implementations and plausibility studies of parts of the story.

This dissertation also has several technical contributions, the most significant of which is a system, called "Ontol", which takes in raw feature-set data (such as the pixels from a collection of image bitmaps) and constructs and uses an ontology for inference and prediction. Ontol serves as a general learner and reasoner and can be used by other systems as long as the other systems can couch their problems as feature-sets. Therefore, we propose that Ontol might serve as the backbone or powerhouse for the bridge and for a fully intelligent system, eventually. In Ontol, the dynamic interplay of 2 relatively simple learning procedures, "chunking" and "merg-

ing", allows us to construct a conceptual structure built of intensions and extensions that allows us to represent and recognize some invariant concepts. This approach is independent of any specific modality. Ontol can also use the same algorithms on previously discovered invariant concepts to create even higher level concepts which will allow for generalizations that aren't easily created without invariant concepts. These generalizations would be analogous to discovering the "<subject> <verb> <object>" pattern in a set a strings generated from a simplified English grammar.

Using Ontol, we make the following technical contributions:

1 An application of Ontol that is a semi-supervised learning system that takes in a large number of unlabeled feature-sets and learns a target concept using a mere handful of labeled training instances. This both demonstrates the utility of the ontology created by Ontol, and is a useful algorithm in itself since it uses fewer training instances than needed by previous systems. Furthermore, we show how this system can learn in the absence of negative evidence.

2 An algorithm that uses ontologies generated by Ontol to improve performance on Reinforcement Learning tasks (Sutton & Barto 1998). We use the concepts learned by Ontol to find useful macro-actions, which can be used used to decrease learning time and increase reward overall for these tasks.

3 An energy function, derived through information theoretic principles, that provides a quantitative measure of the utility for ontology construction and similar models. Ontol searches for conceptual structures that best minimize this function.

Although we consider the core technical contribution of this dissertation to be Ontol, we've also provided simple-yet-concrete "proof of concept" implementations

for some of the other parts of the bridge. The point of these implementations is not to provide a rigorous scientific exploration of these ideas (which would be beyond the scope of this dissertation), but to demonstrate feasibility of our ideas and to provide a starting point for future work. These additional technical contributions are listed below:

4 An implementation of how parameterized functions are called and represented in the same representation framework used by Ontol.

5 A model of analogy that exploits *behavioral* isomorphism (as opposed to structural isomorphism), which is hitherto unexplored. We've provided an implementation for "behavioral signatures" for conceptual structures, and have given a proof-of-concept example that concretizes how these signatures can be used to quickly find analogies for non-pathological cases.

6 A system for representing and discovering transformations, such as color and affine transforms for images or pitch, speed, and "accent" transforms for speech data, without depending on temporal information, and we've shown how these transformations allow us to compress the data set and make inferences on the data.

1.5 A Road Map

The remainder of this document is organized as follows:

Chapter 2 gives a high level description of the full story for how a computer might go from raw feature-set sensor data to a rich relational theory of that data, with notes on which parts are implemented as part of this dissertation.

Chapter 3 describes Ontol, the major technical contribution of this work: a system, loosely based on cortical models, that builds and uses a conceptual ontology using intensions and extensions.

Chapter 4 describes a novel model of analogy that is able to find behaviorally-similar parts of the ontology and use them to better characterize input data.

Chapter 5 describes a novel algorithm that discovers transformations, such as rotation in visual data, and uses these transformations to compress the input data.

Chapter 6 goes into detail about the remaining unimplemented steps for finishing the bridge.

Chapter 7 discusses other missing components for a fully intelligent system, and how these missing components might integrate into the existing story.

Chapter 8 evaluates what conclusions we can draw from the implemented components about intelligence and describes the next steps for achieving this goal.

Chapter 2

SOLUTION OVERVIEW WITH A RUNNING EXAMPLE PROBLEM

This chapter provides an overview of how a system can develop a rich relational theory from raw sensor data. A running example in a simplified vision domain will illustrate some of our assumptions.

2.1 A running example problem

As our running example, we'll use the set of black and white line drawings taken from (George 2008). Each image is a 50 by 50 grid of pixels that are each either ON or OFF. Examples of these images are shown in Figure 2.1. Each image contains one of 181 simple shapes under various rotation and translation transforms, giving us 100,000 total images. From this data set, we want to develop notions such as translation, rotation, and scaling.

The input to our system is a set of features. As shown in Figure 2.2, each pixel is

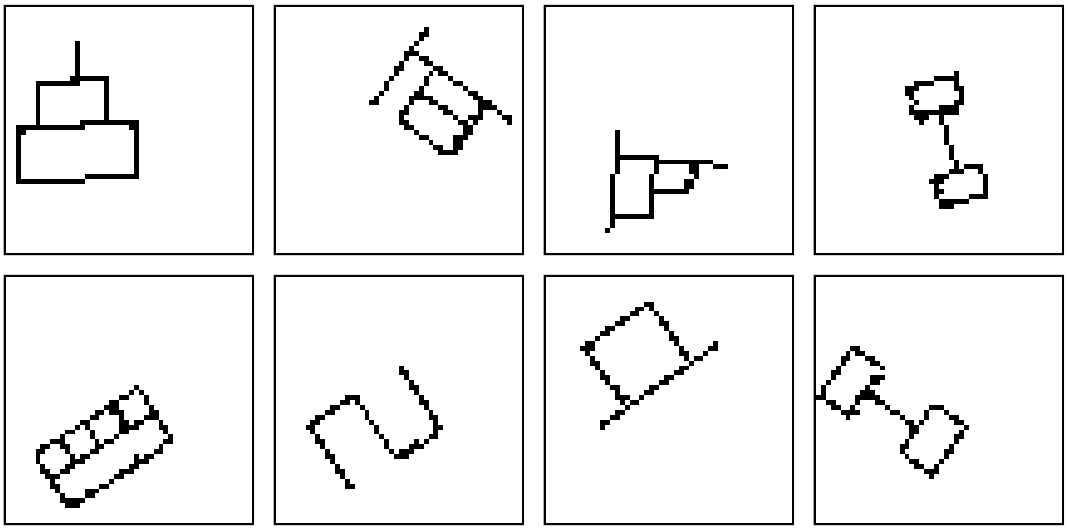


FIG. 2.1. **A sampling of 50 by 50 bitmaps.** Note that the two images on the right are the same shape under different translations and rotations.

a feature, with a value of either "ON", "OFF", or "UNSPECIFIED". These features are given unique names (such as "Sensor2409"). We stress that our algorithm needs to be *free of any innate domain knowledge*. Instead, knowledge about the domain needs to be acquired from massive amounts of data, which we assume we're given. So the features' names are to be treated as gensyms. That is, the feature names themselves provide no information about the features' meanings.

Since the sensor names have no connection to the sensors' meanings, the numbering is in an arbitrary order, so the image in Figure 2.2 is seen as the feature-set in Figure 2.3. At this point, the computer only sees this instance as a set of features. The computer isn't told that the features come from a visual domain, and the computer isn't told that Sensor0351 has anything more to do with Sensor0352 than it does with Sensor2407. The computer's data for this example is a set of 100,000

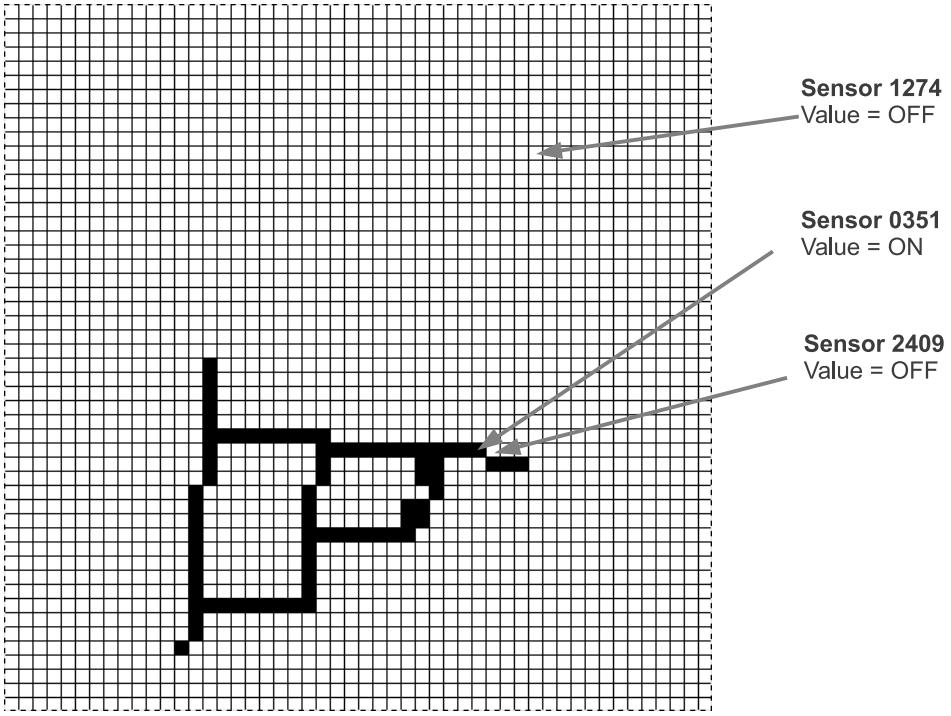


FIG. 2.2. **A sensor grid.** We've arranged the 2,500 sensors in a 50 by 50 grid, though the computer doesn't have this ordering knowledge. In this example, 78 of the sensors are ON (including Sensor0351), and the other 2422 sensors are all OFF. We also allow that a sensor can be "UNSPECIFIED".

{ Sensor0002=ON, Sensor0017=ON, Sensor0048=ON, Sensor0055=ON, Sensor0056=ON, Sensor0117=ON,
Sensor0175=ON, Sensor0180=ON, Sensor0197=ON, Sensor0233=ON, Sensor0269=ON, Sensor0284=ON,
Sensor0341=ON, Sensor0351=ON, Sensor0404=ON, Sensor0444=ON, Sensor0483=ON, Sensor0490=ON,
Sensor0551=ON, Sensor0567=ON, Sensor0573=ON, Sensor0623=ON, Sensor0711=ON, Sensor0729=ON,
Sensor0763=ON, Sensor0779=ON, Sensor0798=ON, Sensor0827=ON, Sensor0833=ON, Sensor0859=ON,
Sensor0947=ON, Sensor0956=ON, Sensor1027=ON, Sensor1043=ON, Sensor1132=ON, Sensor1137=ON,
Sensor1188=ON, Sensor1214=ON, Sensor1239=ON, Sensor1244=ON, Sensor1275=ON, Sensor1305=ON,
Sensor1308=ON, Sensor1352=ON, Sensor1395=ON, Sensor1452=ON, Sensor1555=ON, Sensor1572=ON,
Sensor1579=ON, Sensor1582=ON, Sensor1631=ON, Sensor1651=ON, Sensor1655=ON, Sensor1771=ON,
Sensor1796=ON, Sensor1853=ON, Sensor1891=ON, Sensor1898=ON, Sensor1968=ON, Sensor2059=ON,
Sensor2129=ON, Sensor2137=ON, Sensor2161=ON, Sensor2186=ON, Sensor2195=ON, Sensor2206=ON,
Sensor2214=ON, Sensor2218=ON, Sensor2227=ON, Sensor2247=ON, Sensor2258=ON, Sensor2308=ON,
Sensor2325=ON, Sensor2344=ON, Sensor2355=ON, Sensor2363=ON, Sensor2398=ON, Sensor2406=ON,
Sensor0000=OFF, Sensor0001=OFF, Sensor0003=OFF, etc.... (all 2,419 other sensors are OFF) }

FIG. 2.3. **The raw grid representation as seen by the computer.** The pixels correspond to sensors, which are given arbitrary but fixed numbers.

of these feature-sets (one for each of our 100,000 images). For now, we assume that these 100,000 feature-sets are all given in batch mode, though ultimately we'd like for the system to run on sequential data.

Each image is then an *unordered* set of these features. So, for the computer, the pixels may as well be scrambled in an arbitrary order. To get an idea of the difficulty of the task the computer faces, this scrambling is shown in Figure 2.4. Initially, the computer assumes no relation among the features. The computer doesn't know what the sensors mean, nor know which sensors are close to each other, nor that these are even from a spatial or contiguous domain.

Consider the "dog" image on the top, 2nd from the right in Figure 2.4. The shape in the image to its right is a slight rotation of the dog shape. Though these look similar to us, only 20% of the same Sensors are ON in both of these feature-set. Conversely, nearly half of the ON Sensors in the image to the left of the dog are also ON in the dog image.

Due to these extreme limitations on putting knowledge into our system, it may

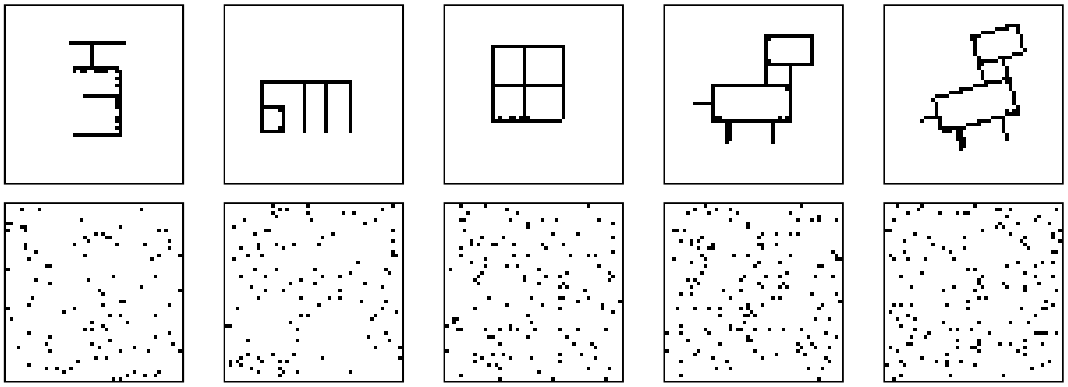


FIG. 2.4. **A representation of the difficulty of the problem.** The computer is not told which pixels are adjacent to which. To get an idea of what this is like for the computer, we've rearranged the pixels for each top image using an arbitrary ordering to get the image below it. The ordering is the same for all images. The computer's problem is actually more difficult: the computer is neither told that its sensors are pixels nor is it given any notion of what a pixel is to begin with. All these concepts must be learned. Note that the two images on the top right are slightly rotated versions of the same shape. Just a slight rotation or translation of a shape will cause a nearly-disjoint set of sensor to turn ON.

seem like this task is rather difficult or even impossible. Indeed, the idea of postulating relations from non-relational data may seem absurd. However, there's some evidence that this is possible.

An initial approach we might use to "unscramble" the sensors is following the ideas of (Pierce & Kuipers 1997). Namely, we can assume a relation between sensors a and b if the correlation between them exceeds some threshold. Doing so on the image dataset yields encouraging results. In the grid shown in Figure 2.5, we've connected each of the sensors (represented as nodes in the graph) with its 8 most highly correlated neighbors. For clarity, we've used a smaller grid: 20 by 20 thresholded image patches taken from natural images.

For this dataset, a node's 4 immediate neighbors (above, below, left, and right) were nearly always among the node's top 8 most highly correlated nodes. And in general, a node's 8 immediate neighbors roughly corresponded to the 8 nodes most highly correlated with it, but there are plenty of exceptions. If we use the same connectivity to layout the graph with an automated graph-layout system, we get a grid-like layout as shown in Figure 2.6.

Though in Figure 2.6 it may seem as though we've gotten back our grid structure, we've actually done this by putting some of our own domain knowledge into the system, which should be discouraged. We set each node's number of neighbors, we've stated that each node has the same number of neighbors, by using the planar graph-layout algorithm we've implicitly stated how many dimensions the space of the nodes is in (two), and we've claimed that these nodes are adjacent or spatially located to begin with. As it is, this approach would break if we had thresholded red, green, and blue values for each pixel instead of just black and white. This "color" approach

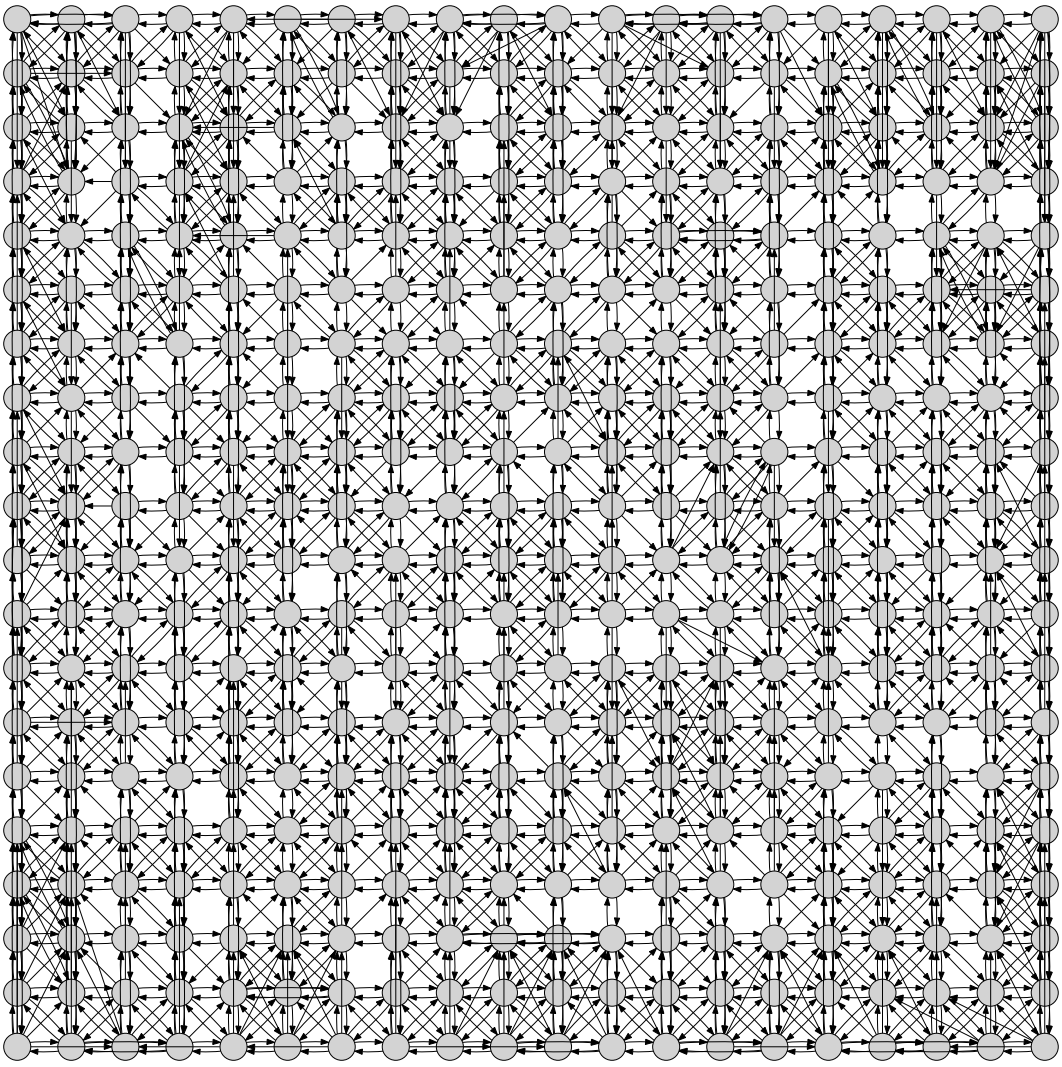


FIG. 2.5. **Postulating relations as correlation on a 20 by 20 grid.** Each sensor is connected to its 8 most highly correlated neighbor. The location for each node was explicitly provided by us. These correlations are taken from a dataset of 20 by 20 patches from thresholded natural images.

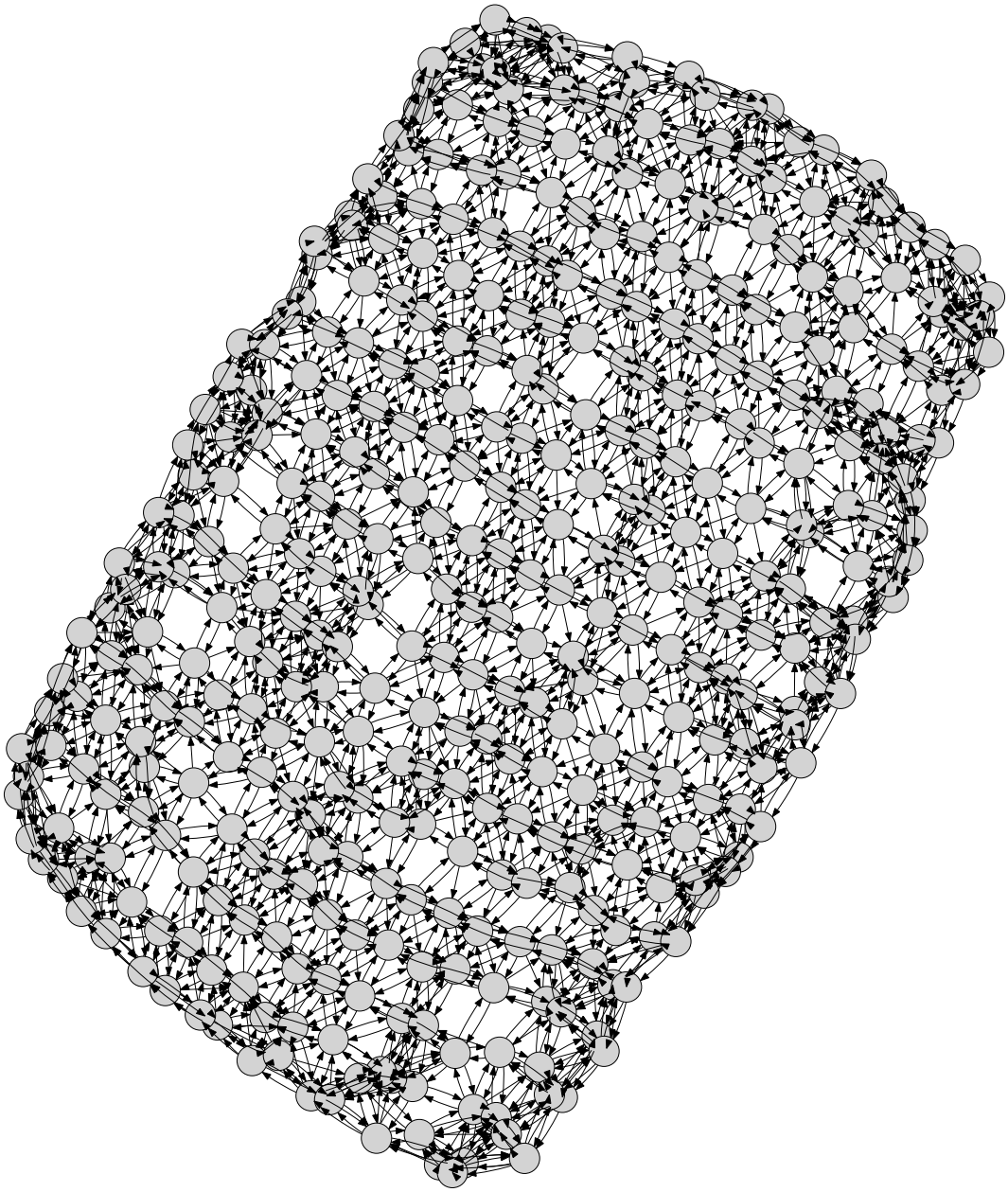


FIG. 2.6. **The "natural" layout using an automated graph layout with the connectivity from Figure 2.5.**

would mean that we have 3 sensor for each pixel.

So we have to be careful. We want the same algorithm to work across a broad range of domains with no modification. Though our running example is from an image domain, we can't apply any image-specific knowledge to our solution. We can't use line segmentation or edge detection, for example. If these features are useful, they need to be discovered by our system.

Not all domains are explicitly spatial or temporal. For example, the card game Poker has no explicit spatial component. Analysis of a static structure, such as a bridge, has no explicit temporal component, and a static analysis of user movie preferences, where we're given what users rated movies (but not *when* they gave those ratings), is neither explicitly spatial nor temporal.

2.2 Solution Overview

The example from the previous section illustrates how our hands are tied. We can't rely on any of our own knowledge about images. We can't even tell the computer which pixels are adjacent to each other. This forces us to find what's common among systems with wildly different modalities. It forces us to simplify our algorithms to their core. This calls for algorithmic elegance, though it's usually more difficult to make something work by simplifying it than by making it more complicated.

In our running example, we'd like to build a system that learns a Theory of 2D Object Rotation. We'd want the system to build the theory being told almost nothing about space (two dimensional or otherwise). We want to build a *bridge*

from raw feature data to a theory that recognizes 2D shapes at different rotations. Ultimately, this theory of rotation that our system develops should include a concept that corresponds to our idea of "angle of rotation". That is, it should parameterize its concept of rotation with a degree. We'd also like our system to develop some notion of a "scalar", and that a degree is a scalar just as is "quantity" if it's characterizing piles of apples.

We propose a bridge consisting of 5 phases:

Phase 1 is a system called "Ontol", which constructs an ontology from a set of feature-sets. The ontology consists of both conjunctions and disjunctions, which some models have shown may be sufficient for representing a large range of invariant concepts (Hawkins & Blakeslee 2004), (George & Hawkins 2009), (Riesenhuber & Poggio 1999). We've implemented the core parts of this phase in this dissertation.

Phase 2 creates *parameterized* concepts by finding behaviorally isomorphic regions of the ontology and parameterizing the differences. In doing this on our image dataset, Phase 2 should also find concepts that are invariant to transformations such as translation, rotation, and scaling. We've implemented some parts of this phase as a proof of concept.

Phase 3 discovers transformations (such as rotating by 45 degrees) by finding consistent mappings among invariant instances of the same concept. We've also implemented parts of this phase as a proof of concept.

Phase 4 then parameterizes *these* transformations by finding consistent properties among several types of relations. For example, quantities of apples are similar to

degrees of rotation in that 57 apples is more similar to 56 apples than it is to 20 apples, just as 57 degrees is more similar to 56 degrees than it is to 20 degrees. Given this, the system should generalize the notion of "quantity" (though it won't know the name) and use this to describe both apples and angles. This phase is still conceptual.

Phase 5 represents relational structures as feature-sets, then uses Ontol on these feature-sets to create and use an ontology of relational structures. We propose this as a basis for metacognition. This phase is also currently conceptual.

Ideally, all 5 of these phases should run concurrently, though this is future work.

2.2.1 Phase 1: Building and Using an Ontology from Feature-Sets

The first phase is a system, called Ontol, that builds an ontology from an input set of feature-sets. In our example, this input set is a collection of 100,000 feature-sets, each of which has 2,500 features. We also allow that not all of the 2,500 features need be specified for each feature-set. If a feature's value is unknown, it's simply omitted. For raw sensor data, this allowance may not be necessary, but data will be unknown for higher level concepts.

Ontol is based on cortical models, particularly the Hierarchical Temporal Memory (HTM) model of (Hawkins & Blakeslee 2004) and the HMax model of (Riesenhuber & Poggio 1999). Both of these models describe connectionist perceptual systems that consist of 2 types of nodes. The ontologies in these systems both consist of what can be thought of as conjunctive and disjunctive nodes. Once built, Ontol uses its ontology in a manner similar to that of HTMs, in that it uses both bottom-up and

top-down reasoning. The specifics of how Ontol builds and uses these structure are fully detailed in chapter 3.

Ontol's ontology is a collection of conjunctive and disjunctive nodes, which we call ANDs and ORs, respectively. These nodes are linked together to form a *heterarchy*, which is a like a hierarchy, but allows that nodes may have multiple parents. We don't allow nodes to point to their ancestors, so the ontology is a directed acyclic graph. We can turn the ontology into a concept lattice by creating a "top" and "bottom" node and pointing the top to nodes without parents, and pointing the leaf nodes to the bottom node.

The goal of Ontol is to *characterize* the input data in an unsupervised manner. Taking a Bayesian approach to model building, Ontol searches for the most probable model given the data. Following (Wolff 2003) and others, we assume a description-length prior, which essentially means that Ontol searches for the shortest model that describes the data. A caveat is that Ontol's model is lossy, so we add $-\log_2 P(D|M)$ to the model's description length, where $P(D|M)$ is the probability of the data D given the model M.

Generally, an OR node is set to ON if any of its children are ON, and if an AND node is ON, then all its children are turned ON (if nothing is keeping them OFF). Nodes can also have inhibitory connections, so if an AND node is on and has an inhibitory connection to a child node, that child node will be OFF. Initially, each node in our observed data is marked as "unexplained". If an AND node has an excitatory connection to an ON but "unexplained" node, then if we set the AND node to ON, the unexplained node will become explained and the AND node will be unexplained. That is, the AND node explains the observed node, but must then be explained itself.

Additionally, AND nodes will "compete" to "explain" their children. If 2 AND nodes both have excitatory connections to a child that's ON but unexplained, then both AND nodes will be swayed to be ON. As soon as one of the AND nodes is set to ON, the child will be explained, and the other AND node will no longer have this influence to be ON. For an example drawn from Bayesian networks, suppose we have a sensor that's ON if and only if a burglar alarm sounds, and the alarm can be caused by both a burglary or an earthquake. Then our network would be 3 nodes: Alarm, Burglary, and Earthquake, where both Burglary and Earthquake would be AND nodes pointing to Alarm, as shown in Figure 2.7. If the alarm sounds, and we know that Earthquake is ON, then Burglary is no longer pushed to be ON to explain the alarm. Note that though Burglary is no longer pushed to be ON to explain the alarm, it's not being inhibited either. Burglary might have other children that need to be explained, so it's conceivable that both Alarm and Burglary are ON.

Conversely, if an OR is ON but none of its children are ON, then the children are pushed to be ON to "instantiate" the OR. As soon as one of the OR's children turn ON, the other children lose this drive to be ON.

ANDs and ORs are simple concepts, but (Hawkins & Blakeslee 2004) and (Riesenhuber & Poggio 1999) argue that these are sufficient for representing sophisticated concepts, such as a good number of invariant concepts. For example, we see the 2 "dog" images at the upper right of 2.4 as basically the same shape. That means that we're able to represent this shape that doesn't change with the shape's rotation. So we have an concept for this shape that's *invariant* with respect to its rotation angle.

Figure 2.8 gives a simplified example of how ANDs and ORs can be used to represent concepts from our visual domain example that are invariant to translation and

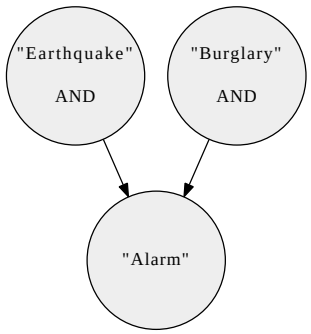


FIG. 2.7. **Earthquake and Burglary both as explanations for Alarm.** If Earthquake is ON, then Burglary is no longer pushed to be ON to explain Alarm.

rotation. In this example, the "dog" concept may take on 2 forms, so it's represented as an OR. Each of these forms is AND, which may be thought of as a bag of features.

Consider the form of "dog" on the right (which is the same as the "dog" shape in Figure 2.4). Suppose we wanted to represent all forms of this shape: all translations, rotations and combinations of these. One approach would be to explicitly list all variants of this shape using a giant OR. To quantify this, suppose we have 24 different rotations $(0°, 15°, 30°, \cdots, 345°)$, and 2,500 translation shifts (recentering the image on each of the 2,500 pixels). This gives 60,000 possible variants for each image. So, using this approach, if we wanted to represent all 181 of our images, we'd have to have list a total of 10,860,000 images. This approach is not only costly memory-wise, but it offers no transfer of knowledge, so it would need to be explicitly shown all 10,980,000 images for its library to be complete. In reality, we'd likely have more than just the 181 sample images, we'd likely have thousands of shapes we'd want to recognize.

Another approach is more efficient with generalization and memory, with a slight

accuracy cost. We consider this accuracy cost to be acceptable, since human vision isn't always accurate as evidenced by various optical illusions and false identification. Instead of memorizing 60,000 variants of each of the 181 shapes, we can have a handful of "basic shapes" each of which we represent invariantly with a 60,000 item OR. That is, we list all 60,000 variants for each of these smaller shapes. This is less taxing for use because it's only a handful of shapes and also because each "basic" shape is smaller than a full high-level shape. Essentially, we "reuse" the basic shapes across multiple images. An example of such an OR is shown as the lower OR in Figure 2.8. Each of the 3 children of the OR shown is an AND that has as its components the specific pixel values for the shown shape in a particular location. For example, the leftmost AND might be the set of 9 pixel values: {$x10y12$=ON, $x10y13$=ON, $x10y14$=ON, $x10y15$=ON, $x10y16$=ON, $x10y17$=ON, $x10y18$=ON, $x09y14$=ON, $x08y14$=ON}. In reality, this OR should have 60,000 children instead of just the 3 shown. Once we have this explicit invariant representation of these low-level features, we can represent invariant higher-level features by saying the parts they consist of. That is, we represent the higher-level features as a bag of lower-level features. We lose information here because for 2 low-level features in a bag, we don't specify *which* instance of one low-level feature goes with which instance of another. We ameliorate this loss of information by ensuring plenty of overlap among the bags.

Our representation uses the insights from the latter approach. The "dog" shape on the right of Figure 2.8 is represented as a bag of features, each of which is the shape part, invariant of translation and rotation. Note that the features have overlap with each other. The dog's children from left to right can be thought of as the "body", the "head", the "neck", and the "tail". Note that nothing explicitly says that the dog's neck overlaps with (or "connects") the dog's body and the dog's head, or that they're

even connected at all. Conceptually, it'd be possible to have a disjoint bag of features that are all ON but clearly don't make up a dog. However, with enough parts in our AND, the visual canvas will be too small to arrange all these parts so that they all fit without some overlap, and, as discussed in (Riesenhuber & Poggio 1999), a great amount of overlap makes it difficult to arrange all the dog parts in anything but a dog. Note that the left and right forms of the "dog" share some features. This feature reuse is fundamentally what allows us to represent invariant concepts efficiently.

Thus, if we know that the right-side dog shape is ON, we don't necessarily know the values for our low-level inputs, but we can turn a configuration for these low-level inputs ON and test whether it causes the dog's bag of features to be active. Thus, the bag of high-level features that is our definition for "dog" effectively works as a one-way "hash" or "signature" for this concept. Unlike checksums, we prefer signature *continuity*. That is, if 2 shapes are similar, then they should have similar hashes.

A problem we plan to explore in future work is searching for low-level configurations to satisfy the high-level signature. If relational structures can be represented as feature-sets, as we argue in Chapter 6, then we can use this search to do planning (among other functions) by encoding actions as low-level features and goals as high-level constraints.

To build this structure, we propose using 2 processes: chunking and merging. Chunking, which we investigate in detail in Chapter 3, finds ANDs by searching for feature-sets that are large and/or occur as subsets of other feature-sets in our ontology. Merging finds ORs by finding sets of features that occur in similar contexts as each other, but rarely occur with each other. We provide a proof-of-concept for Merging in Chapter 3, but we leave a rigorous investigation for future work.

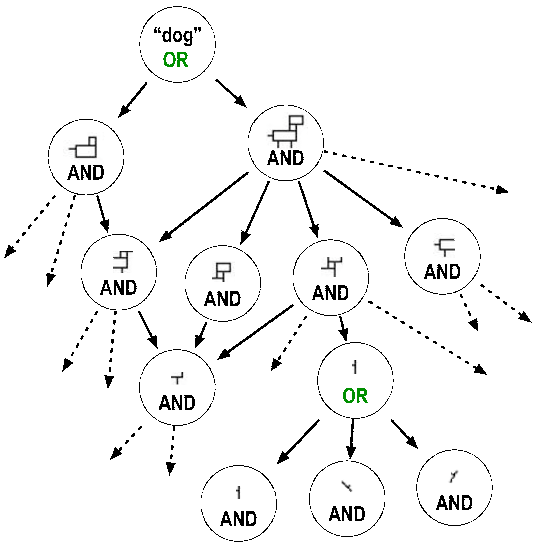


FIG. 2.8. **Representing invariance with extensions ("OR"s) and intensions ("AND"s).**

The chunking algorithm finds concepts that will help compress the data. If a set of features has n elements and occurs m times, we can use it to save $mn - m - n$ links in the ontology by creating a new AND node for this feature-set, having it point to its n elements, then, for every feature-set that has these n elements, remove those n, and instead replace them with a single link to the new AND node. The new node is then a (non-primitive) feature itself. Essentially, chunking is finding feature-set overlap and using it to compress or "explain away" the data. A useful chunk is then turned into a concept and added to the ontology as a feature-set, and this process is repeated. New chunks may include feature-sets from both "original" feature-sets and older "discovered" sets. Thus, chunking builds a multi-level conceptual heterarchy[1].

When chunking is applied to our example bitmap dataset, it comes up with

[1]This is a "heterarchy" as opposed to a hierarchy because a node may inherit from multiple parents. In a strict hierarchy, nodes have only 1 parent, at most.

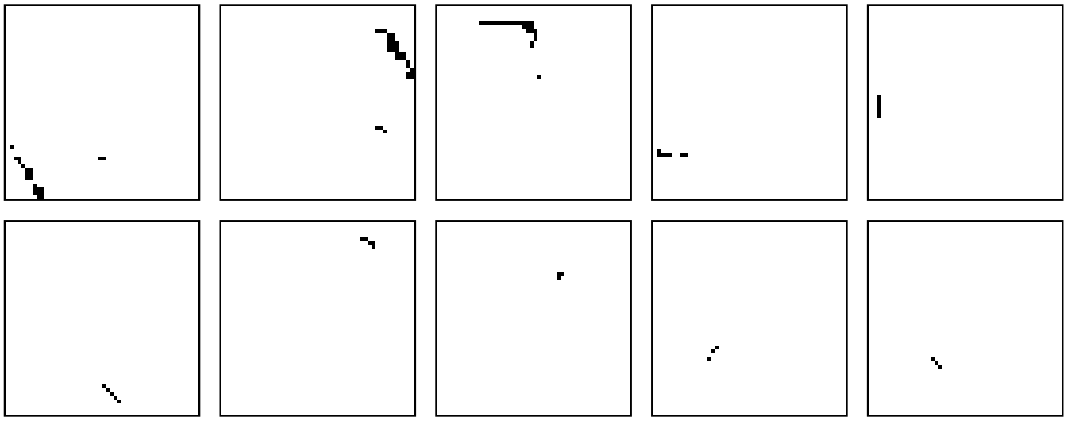


FIG. 2.9. **A sampling of chunks discovered by our "crunching" algorithm.** Concepts are shown in the order they were discovered from top to bottom, left to right. Note that these chunks tend to be clusters of adjacent pixels, though our algorithm was told nothing about the dimensional nature of the pixels. Also, note that *we* have arranged the pixels in a grid to more easily see the chunks.

features such as those shown in Figure 2.9. These feature-sets tend to be contiguous groups of adjacent pixels. So chunking was able to find some idea of pixel contiguity with nothing specific to vision encoded in the algorithm. As will be shown in Chapter 3, this same algorithm can be used on virtually any binary feature-set.

When we apply chunking alone to a small set of 183 20 by 20 patches randomly taken from natural images, we get the ontology shown in Figure 2.10, which gives an idea of the structure of the heterarchy found. (The ontology found from the bitmap data was too large to be illustrative.)

Merging works by finding features that are interchangeable. These features can be primitive features or higher-level "created" features. For features or concepts to be interchangeable, we need to consider the *context* under which they're interchangeable. For example the words "dog" and "memory" don't have much in common in terms

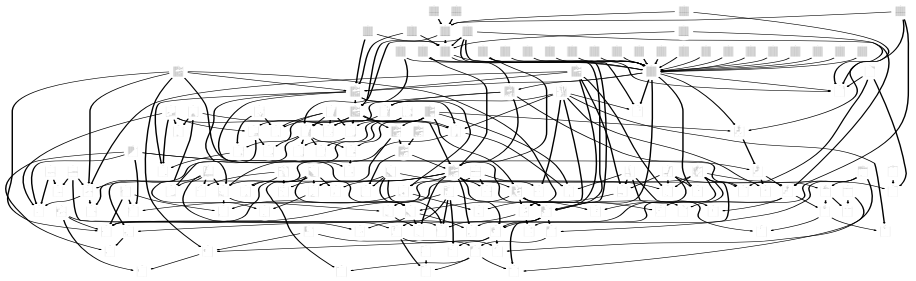


FIG. 2.10. **An ontology created by "chunking" a set of 20 by 20 patches taken from natural images.** Note that the chunks found tend to be contiguous sets of pixels. Arrows all point downward and represent inheritance. Each node is an AND and is shown as the "flattened" image of the nodes from which it inherits. For this example, we only specified a pixel ON if it's black. White pixels were all left UNSPECIFIED. The original feature-sets are identifiable as the AND nodes with no parents. Not shown are the 35 all-white patches, which would be grouped with the nodes on the right that have no parents.

of their semantics, but both are nouns and both can play the same syntactic role in sentence construction. Likewise an aluminum baseball bat and a soda can are the same in terms of which recycling bin to put them in, but different with respect to their utility for hitting baseballs. When a set of concepts are found that are interchangeable, we "merge" them, creating an equivalence class with these items. We then use this equivalence class as an OR node in our ontology.

Merging allows us to more easily find useful chunks. For example, the "`<noun>`, `<verb>`, `<noun>`" construction in English wouldn't be as easily visible without the concept of `<noun>`s and `<verbs>`s. Another example is shown in Figure 2.11, where routes along a commute are shown to be equivalent under a certain context. Once these equivalence classes are found, we can find patterns such as "Tim is at home. Tim takes *a work-bound route*. Tim is at work. Tim takes *a home-bound route*. Tim is at home.".

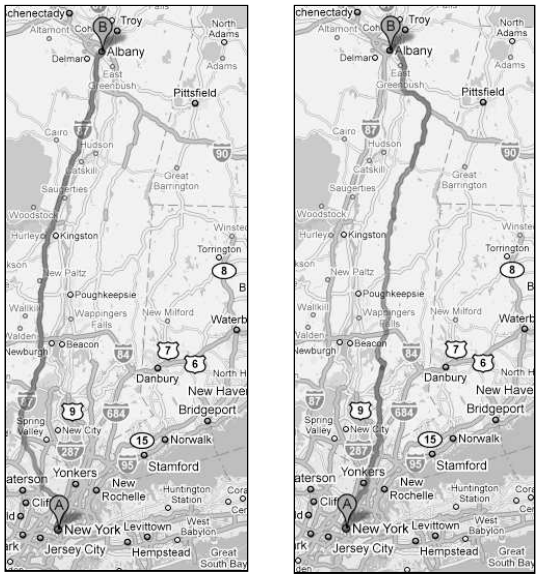


FIG. 2.11. **"Equivalent" routes from New York City to Albany.** The image on the left shows the route taking Interstate 87 and the route via the Taconic State Parkway on the right. There are many differences between the routes, to be sure. The route on the left passes near Stewart International Airport, for example. But both routes form an equivalence class *in the context of* ways to drive from New Work to Albany in roughly 3 hours.

Merging comes at a cost, though: it is lossy, and decreases the probability of getting the data from the ontology. This lossiness is balanced by the ability we gain to form new concepts. Ultimately, merging will allow us to compress the input data, which we will discuss in Chapter 3.

Currently, only Chunking is fully implemented, but in future work, Ontol will interleave these 2 process —chunking and merging— to create an ontology consisting of ANDs and ORs like that in 2.8.

Although the AND/OR representation we use is sufficient for representing a "dog" invariant of its rotation, translation, and scale, chunking and merging by them-

selves are insufficient for discovering this invariance.

To see this, consider translation invariance. Suppose we have a set of 100 images of identical dog shapes, except each image is the shape translated to a different place on the grid. Further suppose that chunking has carved up each dog into a head, a neck, and a body. At this point, we'll have 10 instances of "dog head", say `doghead1`, `doghead2`, $\cdots$, `doghead10`. Note that these names are for our convenience for this example, but they have no meaning for our system. These names may as well be `Gensym-XQ41J`, `Gensym-P87UF`, etc.. So we don't know that these `dogheads` have anything to do with each other. None of the `dogheads` occur in the same context: `doghead1` occurs with `dogbody1` and `dogneck1`, while `doghead2` occurs with `dogbody2` and `dogneck2`.

If we knew that `doghead1`, $\cdots$, `doghead10` formed an equivalence class (call it `DOGHEAD`), then merging would recognize that `dogneck1`, $\cdots$, `dogneck10` and `dogbody1`, $\cdots$, `dogbody10` all occurred interchangeably in the same context as `DOGHEAD`, and may form an equivalence class with all these. Note that, at this point `dogbody1` and `dogneck1` are symmetrical, so we still have no way of creating the equivalence class for `DOGBODY`. If the `dogbodys` all occurred in some other context (such as with `WOLFHEAD`), then this would break the symmetry, and merging would form the equivalence classes `DOGBODY` and `DOGNECK`.

This all begs the question of where we get `DOGHEAD` and `WOLFHEAD` to begin with. Both occur in the occur in the context of `DOGBODY`, and vice versa, but we need to find one of these equivalence classes initially. To do this, we need to look for patterns in the *relations* among these features. This is where the next phase, Parameterized Concepts, comes in.

2.2.2 Phase 2: Parameterized Concepts

Once Ontol has created an ontology with our data, we may notice that different parts of the ontology are basically isomorphic. For example, the ontology that covers the 10 by 10 square in the upper right of our bitmap sensor grid from Figure 2.2 will behave a lot like the 10 by 10 square in the lower left. So we have what's basically the same functionality repeated twice (many times actually). If we could extract out this functional overlap, not only would we decrease our description length, but we'd get other benefits such as generalization and knowledge transfer. Essentially, this is an *analogy* from one cortical region to another. Analogy has been shown to have many cognitive benefits by (Hofstadter 2001), (Forbus 2001), (Dietrich 2000), to name a few.

This extraction is what Phase 2 does. Phase 2 does analogical schema induction by finding *behavioral* overlap among cortical regions (i.e., cortical regions that behavior similarly to each other), forming a new concept by extracting out the similarity, and parameterizing the differences. That is, where the cortical regions differ, we create slots that can have various fillers. These fillers then form an equivalent class. For example, if we see the following situations (and others like them):

Martha has a pen.

1. Doug takes the pen.

   Martha no longer has the pen.

A black ant has a beetle corpse.

2. A red ant takes the beetle corpse.

   The black ant no longer has the beetle corpse.

A company has a corporate secret.

3. A corporate spy takes the corporate secret.

   The company no longer has the corporate secret. (It's no longer a secret.)

we might create the concept "`steal`", which has the following definition

Z has **Y**.

X takes **Y**.

Z no longer has **Y**.

where **X**, **Y**, and **Z** are slots or parameters for this concept. Then we can encode our 3 cases using this new concept:

steal (Doug, pen, Martha)

steal (corporate spy, corporate secret, company)

steal (red ant, beetle corpse, black ant)

Now we form equivalence classes with the fillers

$\mathbf{X}$ = OR (Doug, red ant, corporate spy, $\cdots$)

$\mathbf{Y}$ = OR (pen, beetle corps, factory plans, $\cdots$)

$\mathbf{Z}$ = OR (Martha, black ant, company, $\cdots$)

Thus, Doug, red ant, and corporate spy (among others) form an equivalence class because they can fill the slot **X**. We might also note that these share the *intensional* quality that they're all animate.

To see how this might be applied to our bitmap grid example, suppose chunking and merging gives us the following sets:

{{doghead1, dogbody1, dogneck1}, {doghead2, dogbody2, dogneck2},
{doghead1, a1, b1}, {doghead2, a2, b2},
{dogneck1, b1, c1}, {dogneck2, b2, c2},
{dogbody1, a1, c1}, {dogbody1, a2, c2},

{doghead3, dogbody3, dogneck3}, {doghead4, dogbody4, dogneck4},
{doghead3, a3, b3}, {doghead4, a4, b4},
{dogneck3, b3, c3}, {dogneck4, b4, c4},
{dogbody3, a3, c3}, {dogbody1, a4, c4}}

If we create the definition for a new parameterized concept M, where M is shorthand for the set of sets below,

{{**H**, **B**, **N**},
{**H**, **X**, **Y**},
{**N**, **Y**, **Z**},
{**B**, **X**, **Z**}}

where **H**, **B**, **N**, **X**, **Y**, and **Z** are all variables, then we can use M to *compress* our original 16 statements into just 4 statements.

M(**H**=doghead1, **B**=dogbody1, **N**=dogneck1, **X**=a1, **Y**=b1, **Z**=c1)
M(**H**=doghead2, **B**=dogbody2, **N**=dogneck2, **X**=a2, **Y**=b2, **Z**=c2)
M(**H**=doghead3, **B**=dogbody3, **N**=dogneck3, **X**=a3, **Y**=b3, **Z**=c3)
M(**H**=doghead4, **B**=dogbody4, **N**=dogneck4, **X**=a4, **Y**=b4, **Z**=c4)

Note that the parameters that can fill in **H**, namely doghead1, doghead2, doghead3,

and `doghead4`, then form an equivalence class (as do the parameters the can fill in **B**, **N**, **X**, **Y**, and **Z**).

The example we've shown here is filled with simplifying assumptions, but provides the basic idea for how we can get translation invariance. As we suggest later, once these assumptions are addressed, this approach can also give us rotation invariance, scale invariance, and other transform invariances.

When searching for concepts, such as M, we use the same principle as for Ontol, namely we search for concepts that allow us to compress the data.

A problem with this approach is that we assume that each instance of the dog is carved up in a similar way. In reality, this is rarely the case. Furthermore, we want to find subgraphs in our ontology that have similar *behavior* as how they make inferences (using Ontol's inference algorithm). This is because an ontology can be thought of as a Boolean expression, a series of conjunctions and disjunctions. For any truth-table, there are many forms of a Boolean expression that will also have this truth-table. Note that if 2 regions are structurally isomorphic, they will also behave the same, so Phase 2, when fully implemented, will find regular analogy in addition to "behavioral analogies".

Determining the equivalence of 2 Boolean expressions is NP-hard. This is because if we had a polynomial-time algorithm for determining equivalence of a Boolean expressions, then we could use it to solve 3-SAT in polynomial-time simply by determining whether the 3-SAT instance was equivalent to "False". But we're only interested if 2 cortical regions behave *similarly*. Their behavior doesn't have to be completely isomorphic. With this in mind, we use "behavioral signatures" (described

in Chapter 4) to quickly determine whether 2 cortical regions are behaviorally isomorphic. We have implemented behavioral signatures as a proof-of-concept, but we leave a rigorous investigation of these as future work.

Finding behaviorally isomorphic (or nearly isomorphic) regions gives us all the benefits we get from analogy use and discovery: generalization, lower description length, transfer of knowledge, etc. as discussed in (Hofstadter 2001) and elsewhere.

For our bitmap grid example from Figure 2.2, note that if some cortical region "A" is behaviorally isomorphic to some other cortical region "B", then A will also be behaviorally isomorphic to B rotated 90 degrees. If we go up the heterarchy a few levels, cortical region A will also be behaviorally isomorphic to cortical region B rotated 15 degrees. Also, in terms of scale, cortical region A will behave like the cortical area for our entire grid if we go up several levels in this latter. This gives an idea for how Phase 2 gives us scale invariance. Further details of this approach are given in Chapter 4.

2.2.3 Phase 3: Discovery of Mappings

By finding behaviorally isomorphic regions of our ontology, we'll recognize that one "dog" shape is the same as another version of the same shape rotated 45 degrees, but we won't have generalized the idea of rotating a shape 45 degrees. That is, we won't recognize that the *relation* between a "dog" and the rotated dog is the same as the relation between a "duck" shape and the same shape rotated 45 degrees.

Thus, it would be useful to discover a set of mappings. We assume that we have a collection of feature-sets that, through Phase 2, we know to all form an equivalence

class. For example, we recognize that

{doghead1, dogbody1, dogneck1}

{doghead2, dogbody2, dogneck2}

{doghead3, dogbody3, dogneck3}

{doghead4, dogbody4, dogneck4}

are all instances of the "same" shape. Furthermore, we know that

{cathead1, catbody1, catneck1}

{cathead2, catbody2, catneck2}

{cathead3, catbody3, catneck3}

{cathead4, catbody4, catneck4}

are all instances of the same shape and likewise for instances of each of our 181 basic shapes. At this point, we should point out that there will be a good deal of overlap among our features. So a more realistic example for our shapes might look more like the following:

Shape 1	{090.15, 090.05, 000.10, 000.10, 000.10, 000.15}
	{045.15, 135.05, 045.10, 045.10, 045.10, 135.15}
	{060.10, 060.10, 060.10, 060.15, 150.15, 150.05}
Shape 2	{000.25, 090.15, 090.15, 090.15, 090.15, 090.05, 000.05, 000.05}
	{045.25, 135.05, 135.15, 135.15, 135.15, 135.15, 045.05, 045.05}
	{060.05, 060.05, 150.15, 150.15, 150.15, 150.15, 060.25, 150.05}
Shape 3	{000.25, 000.05, 000.05, 000.10, 000.10, 090.25, 090.25, 090.10}
	{045.25, 045.10, 045.10, 045.05, 045.05, 135.10, 135.25, 135.25}
	{060.05, 060.05, 060.10, 060.10, 150.25, 150.25, 150.10, 060.25}
Shape 4	{000.25, 000.25, 090.15, 000.10, 090.25, 090.10}
	{045.25, 045.25, 045.10, 135.15, 135.10, 135.25}
	{150.15, 150.25, 150.10, 060.25, 060.25, 060.10}

In this hand constructed example, the feature "090.25" means that the shape has a line that has rotation angle of 90 degrees and length from between 20 and 25 pixels. In reality, our results from Phase 2 won't be as easily interpretable. Note that these aren't proper sets, but rather "bags", since features can occur more than once, as "000.10" does in our first set. (This means that this shape has 3 lines of length 10 at a 0 degree orientation.) Here we're told that the first 3 feature-sets are all instances of Shape 1, but these sets are in no particular order for our system. We know that the 1st is from the "non-rotated" version of the shape, the 2nd is the shape rotated 45 degrees, and the 3rd is the shape rotated 60 degrees, but our system isn't given this information. In this example, there's a 1 to 1 correspondence between a feature and

the "rotated" version of that feature in the "rotated" feature-set. In reality, there's more noise than this. These issues are further addressed in Chapter 5.

We can then compress the data by using the following mappings

Mapping A			**Mapping** B		
000.05	$\rightarrow$	045.05	000.05	$\rightarrow$	060.05
000.10	$\rightarrow$	045.10	000.10	$\rightarrow$	060.10
000.15	$\rightarrow$	045.15	000.15	$\rightarrow$	060.15
000.25	$\rightarrow$	045.25	000.25	$\rightarrow$	060.25
090.05	$\rightarrow$	135.05	090.05	$\rightarrow$	150.05
090.10	$\rightarrow$	135.10	090.10	$\rightarrow$	150.10
090.15	$\rightarrow$	135.15	090.15	$\rightarrow$	150.15
090.25	$\rightarrow$	135.25	090.25	$\rightarrow$	150.25

These mappings are applied to a feature-set, and (using mapping A) if the feature 000.05 appears in the feature-set, it replaces it with 045.05. For example, if

$$S_1 = \{090.15,\ 090.05,\ 000.10,\ 000.10,\ 000.10,\ 000.15\}$$

then we can apply mapping A to S_1 (by calling $A(S_1)$), which maps the components as shown below.

$$\begin{array}{rcccccc} S_1 = & \{090.15, & 090.05, & 000.10, & 000.10, & 000.10, & 000.15\} \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ A(S_1) = & \{135.15, & 135.05, & 045.10, & 045.10, & 045.10, & 045.15\} \end{array}$$

Thus, $A(S_1)$ is identical to the 2nd instance of Shape 1. So, we can simply say

$A(S_1)$ instead of listing out all the components of the 2nd instance of Shape 1. So we can apply these mappings to express our data as

$$\text{Shape 1} \quad \begin{array}{l} S_1 = \{090.15,\ 090.05,\ 000.10,\ 000.10,\ 000.10,\ 000.15\} \\ A(S_1) \\ B(S_1) \end{array}$$

and similarly for Shapes 2-4. Thus, we need only express the mappings and an instance of each shape.

This is what Phase 3 does: finds mappings, such as those shown here to compress the data. Note that mapping A corresponds to rotating a shape 45 degrees and mapping B corresponds to rotating a shape 60 degrees.

At this point in our story, Phase 3 will have discovered these transformations, but we will not have related them to each other. At this stage, our system won't recognize that mapping B has anything more to do with mapping A than it would to a mapping that essentially translates the image up 20 pixels and right 35 pixels. Discovering relations among these mappings is addressed in Phase 4.

2.2.4 Phases 4 and 5: Parameterizing Mappings

The next 2 steps, Phases 4 and 5, are entirely conceptual, but are provided in outline to complete the story of how we might develop rich relational representations from raw feature sets. Full implementations of these ideas are left as future work.

Given a set of mappings and a set of feature-sets over which these mappings

operate (such as are provided from Phase 3), Phase 4 attempts to discover relations among these mappings and parameterize the mappings if possible. When applied to our grid example, Phase 4 should give us a generalized theory of rotation, translation, and scale, with some idea of a rotation angle.

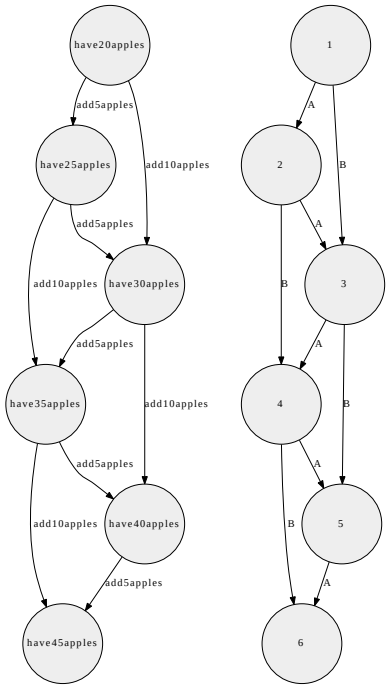
Phase 4 randomly applies the set of mappings to items in the set of feature-sets and constructs a graph where the edges are mapping names and the nodes are names of feature-sets. An edge is drawn from node a to node b (tagged with mapping name x) if mapping x applied to feature-set a produces feature-set b.

Given this graph, we might notice that applying mapping x to a feature-set usually produces the same result as applying mapping y then z to a feature-set.

We might also notice that this graph is isomorphic to another graph with completely different nodes and edges. For example, we might have a graph concerning transformations on apples. In this "apple" graph, we can add or remove apples. So applying the add5apples to have20apples produces have25apples. Then applying the add5apples again to have25apples produces have30apples. Likewise, applying add10apples to have20apples produces have30apples. From other examples like this, we conclude that applying add5apples twice is roughly the same as applying add10apples once.

An example of the graphs that we come up by applying these mappings is shown in Figure 2.12. Note that these graphs are isomorphic. The next step is to extract out the commonalities of these graphs, and parameterize the differences. The parameters that can fill these differences form an equivalence class. We then *Ontolize* these parameters. That is, the parameters that can fill a slot is a set of feature-sets, we can

feed these feature-sets into Ontol to more concisely characterize them, find patterns on them, and do inference with them. This ontology of the parameters is now an intuitive theory of integers.



(a) Apples (b) Angles

FIG. 2.12. **The graphs for Apples and Angles.**

For Phase 4, we can use the same "extract and parameterize" algorithm that we used in Phase 2. In Phase 2, we can also use *intensions* (or a combo of intensions and extensions) to characterize the parameters. Thus we build an ontology of these parameters.

Finally, Phase 5 is the MacGlashan transform, which represents relational structures *as feature-sets*. So we can encode structures such as those in Figure 2.12 as

feature-sets, then use Ontol to find patterns, do inference, and search on these.

These ideas are further discussed in Chapter 6.

Chapter 3

ONTOL: A FEATURE-SET ONTOLOGY

To build our bridge from raw sensors to a rich theory of the world, we must first discover relations among our features. To do so, we build a conceptual structure. In this chapter we describe Ontol, a system that takes in a collection of feature-sets and builds and uses an ontology to characterize these feature-sets. An overview of Ontol is given in Subsection 2.2.1. Once an ontology is built, the model is used for prediction and characterization. This is done through a number of *parse* and *truth-derivation* operators on the ontology, which are described in Section 3.4.

3.1 Related Work

Here we review related work on autonomous ontology formation and unsupervised learning from feature-sets, as well as work on representing and using ontologies to do inference on feature-sets.

A popular framework for learning with and using feature-sets is Bayesian Net-

works (described in (Pearl 1985)), in which every node has a conditional probability table for all the 2^n possibilities for each of its n neighbors (assuming binary variables). A problem with using a Bayesian Network framework is that, in the standard version, the size of a node's conditional probability table is exponential in the number of other nodes it's connected to. I want to allow that some nodes might be connected to thousands of other nodes. Thus, a generic Bayesian framework is impractical. Furthermore, building Bayesian networks (including finding hidden nodes) is expensive. We want to be able to handle massive amounts of data of high dimensionality. Inspired by neurological models, we're only interested in a special subset of Bayesian networks that we hypothesize will be useful for characterizing real-world data, and we can leverage properties of these networks to allow for efficient representation and computation with them. Effectively both Bayesian networks and Ontol have the same expressive power in a "Turing-complete" sense, that of propositional logic. But, by changing the hypothesis space, we can bias our search so that we can more easily find structures that more naturally fit real-world systems, as argued in (Hawkins & Blakeslee 2004).

This work is inspired by and loosely based on models of the neocortex. In particular, this work borrows heavily from the representation of (Riesenhuber & Poggio 1999) and (Hawkins & Blakeslee 2004), and the inference models of (Hawkins & Blakeslee 2004) and (George & Hawkins 2009). Though these models provide insight into how feature-set ontologies can be represented and used, neither of these models provide a satisfactory account of how the ontologies are learned from raw data in an entirely autonomous fashion. Though George and Hawkins give an algorithm for learning weights for a cortical region given a structure, this structure must be hand-built. Likewise, HMax (Riesenhuber & Poggio 1999) gives no learning algorithm for

learning the structure of its model.

HMax (Riesenhuber & Poggio 1999) demonstrates that, by using 2 types of network nodes, essentially AND nodes and OR nodes, visual concepts can be represented and recognized invariantly of rotation and other transformations. Essentially, HMax represents a shape as a bag of parts, as the invariant "dog" shape is represented in Figure 2.8, and uses the same principles that we use to represent shapes with invariance to transforms, as we describe in Subsection 2.2.1. That is, ORs allow us to represent invariance and ANDs allow for efficient encoding. HMax also demonstrates that, with part overlap, it's difficult to find a shape that has all the parts in our bag-of-parts representation for a shape, but is not the shape itself.

Hierarchical Temporal Memories or HTMs ((Hawkins & Blakeslee 2004), (George & Hawkins 2009)) go further and show how this representation can be used for not only bottom-up recognition, but also top-down prediction, which Ontol uses. There are minor differences in the truth-derivation algorithms of Ontol and HTM. For example, HTM seeks to model human cognition, whereas Ontol seeks to attain an information-theoretic ideal. This constrains HTM from an Artificial Intelligence perspective since some constraints, such as the inability to memorize all of one's sensor stream, don't necessarily apply to computers. Ontol also allows for hypothetical branching, where the truth-state of the ontology can be stored, a "hypothetical situation" postulated and its implications tested before "resetting" the ontology back to its original state.

The main difference between Ontol and HTMs is that there is currently no algorithm for creating the *structure* of these memory systems. HTMs require that a user provides this structure. Once a structure is provided, current HTM algorithms can

learn the parameters (e.g., link weights) for that structure from data. This is a problem that this dissertation addresses through the processes of chunking and merging, which are outlined in Subsection 2.2.1 and detailed in this chapter.

3.1.1 Chunking

Many of the papers that give a computational account of concept formation do so in terms of clustering (Thompson & Langley 1991), (Cohen 1998), (Rosenstein & Cohen 1999), (Iba & Langley 2001). Clustering is perhaps the most common form of unsupervised concept formation. For an overview of some of these algorithms, see (Fasulo 1999). Although clustering is useful for some domains, and many of the principles of clustering can be applied to concept formation in general, these techniques alone are inadequate for formation of "analogical" concepts such as the RISK "Border Escalation" described in Subsection 7.1.3, because, to begin with, it's difficult to represent even a single instance of a Border Escalation in a form that a clustering algorithm could use (such as a vector). Second, clustering algorithms usually require the data to be segmented into instances. Another drawback to much of the clustering work is that each item belongs in only one cluster. Hierarchical Clustering (such as (Olson 1995)) allows that an item may be in nested clusters, but usually allows the item to have only 1 parent. Our chunking algorithm, "The Cruncher", uses multiple inheritance, which allows it to overcome obstacles faced by strictly hierarchical classification systems such as hierarchical clustering and decision trees. For example, in Figure 3.7, The Cruncher describes the Penguin (which has attributes in common with Birds and Fish) as both a "bird" and an "aquatic" creature. Hierarchical clustering would force the Penguin to be in only one of these classes. Fuzzy Clustering

(Hoppner *et al.* 1999) has been proposed to address this issue, but these techniques generally require a distance metric to be provided. We take a different approach that doesn't require a user to provide a domain specific distance metric. Kanerva (Kanerva 1988) uses an exemplar model to represent and retrieve large bit-vectors, but doesn't build a concept hierarchy.

There has also been work on Ontology Formation (for example, see (Gruber 1993)), but much of this work is aimed at knowledge engineering, where a human's help is required to assist the ontology formation, or is applicable to a narrow domain. The Cruncher is completely unsupervised in contrast, and counts domain independence as one of its strong points. Many of the ontology generating systems such as OntoLearn (Velardi, Fabriani, & Missikoff 2001), and OntoBuilder (Gal, Modica, & Jamil 2004) are geared specifically for extracting ontologies from written natural language text (employing engineering techniques such as text parsers with hand designed rules) or forms filled by a human user. The Cruncher also allows for exceptions, which further sets it apart from most of the work in this community. The Cruncher's exceptions allow for better compression, and, for the UCI Zoo Database, allow for classifications that correspond to the human-developed classification. For example, the Platypus is described as an egg laying mammal, even though mammals are defined as not laying eggs.

The field of Formal Concept Analysis provides methods for producing Concept Lattices, which form an ontology with multiple inheritance. The input to these algorithms is a matrix where the rows are items and the columns are features of those items. These algorithms then produce a concept lattice where upward links represent feature inheritance. The main aspects that set The Cruncher apart from this

work are: first due to the rigidity of Formal Concept Analysis, exceptions are not allowed as they are for The Cruncher, and second, MDL is not a driving factor in producing the concept lattices. Furthermore, in (Pickett & Oates 2005) we show that The Cruncher can provide better compression than standard Concept Lattice layout algorithms, such as that described in (Cole 2001). For an overview of the field of Formal Concept Analysis see (Ganter & Wille 1999).

The idea for The Cruncher grew out of "PolicyBlocks", an algorithm for finding useful macro-actions in Reinforcement Learning (Pickett and Barto (Pickett & Barto 2002)). The Cruncher extends PolicyBlocks by framing it in terms of ontology formation and MDL, and by adding exceptions and the ability to create multiple levels of concepts.

Grammatical Inference There has been substantial work on chunking in grammatical inference. For a survey of some of this work, see (Gobet *et al.* 2001). Most of these algorithms search for "chunks" which are n-grams that allow us to concisely express the strings (Servan-schreiber & Anderson 1990). For example, ADIOS (Edelman *et al.* 2004) builds a tree of chunks, but this work doesn't have a metric for choosing good chunks grounded in information theory. The MDLChunker algorithm (Robinet & Lemaire 2009) uses an energy function grounded in information theory, like our Equation 3.1, but is also geared toward sequential data. None of these algorithms work by explicitly finding intersections among existing chunks. Furthermore, most *depend* on the sequential information, that we are trying to discover in the first place. Therefore, none of these algorithms would be applicable to undifferentiated sensor data, such as our example in Figure 2.4. Additionally, these algorithms construct *hierarchies*, not heterarchies, and don't allow for exceptions, as we do. The

structures created by these algorithms are grammars, and have no model for inference *using* the structures akin to our inference algorithm in Subsection 3.4, which is based on the cortical models of (Hawkins & Blakeslee 2004) and (Riesenhuber & Poggio 1999).

There has also been work on finding chunks by segmenting strings, such as (Batchelder 2002), (Perruchet & Vintner 1998), and (Armstrong & Oates 2007), but these also suffer from the constraint that they depend on sequential knowledge.

3.1.2 Merging

Merging is the process of finding equivalence classes by finding sets of features that occur in similar contexts as each other, but rarely occur with each other. We call these equivalence classes "OR"s, and they are necessary for our representation model.

Less work has been done on merging than on chunking. Nearly all of this work has been for grammars or minimizing finite state automata. Generally, these methods merge states or grammar rules, which decreases description length at a cost of accuracy. This is the basic approach taken by Bayesian Model Merging (Stolcke & Omohundro 1994). We also take this general approach, but our approach is different in that it looks for features that tend to share equivalent contexts.

3.2 Representation

> What is laid down, ordered, factual, is never enough to embrace the whole truth.
>
> –Boris Pasternak (1890-1960)

Representing and learning invariant concepts is an essential component of intelligence. For example, we can learn to recognize words in our native language largely invariant of the words' pitch, speed, or accent. People have shown that we can represent concepts that are invariant to many transformations (e.g., translation, rotation, and scale for vision) using layers of intensions and extensions (Hawkins & Blakeslee 2004), (Riesenhuber & Poggio 1999), but none have good story for how these structures are built: both include such a degree of human knowledge that their construction algorithms aren't applicable to undifferentiated sensor data.

We want a representation framework that's simple (as argued in our "Desiderata" in Subsection 1.2.1). Yet, if this is to be our core representation framework for a system that's able to plan, we would like our framework to be expressive enough to represent propositional logic. At the same time, to constrain our learning process, we want our representation to be a natural "fit" to the real world. As argued in (Hawkins & Blakeslee 2004), the natural world naturally fits a hierarchical structure.

Our entire ontology is a set of feature-sets, where each set is designated an AND node or an OR node (as done by HMax and HTMs and as shown in Figure 2.8). Every node is given a unique name, and nodes may point to other nodes, excepting that a node may not point to its ancestors. (That is, we don't allow cycles.)

Note that our entire ontology can be represented in a single text file, with one node per line. For example a pair of lines from our ontology might look like this:

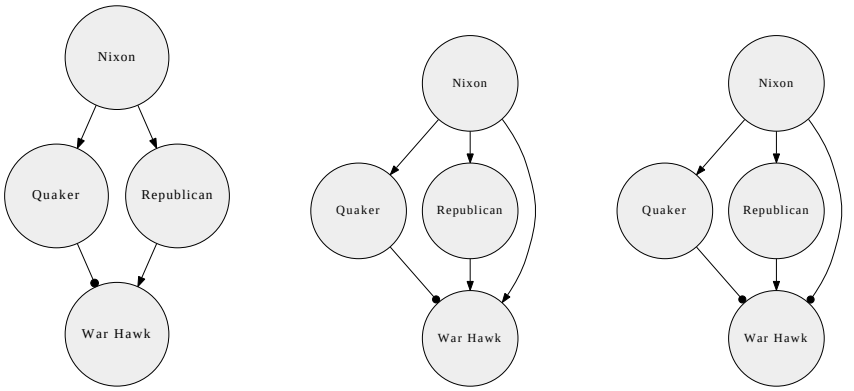
```
node121721: OR {node12132, Sensor1102, Sensor0021, node7382, NOT node73811}
node121728: AND {Sensor1712, Sensor0021, Sensor0021, NOT Sensor0002, Sensor2037}
```

This describes `node121721` as an OR node that points to nodes `node12132`, `Sensor1102`, `Sensor0021`, `node7382`, and `node73811`. This means that node121721 turns ON if any of nodes `node12132`, `Sensor1102`, `Sensor0021`, `node7382` are ON or node `node7382` is OFF. Conversely, if the OR node `node121721` is ON, then at least *one* of its children are driven to be ON (except for `node73811`, which might be driven to be OFF) to satisfy the OR property. (Nodes `Sensor0021` and `Sensor001102` happens to be a raw sensor node, but the name is for our convenience.) More details of this process are given in Section 3.4.

The 2nd line describes `node121728`, an AND node the points to nodes `Sensor1712`, `Sensor0021`, `Sensor0002`, and `Sensor2037`. If `node121728` is ON, then these nodes are all driven to be ON, except for node `Sensor0002`, which will be driven to be OFF, since the link to it is inhibitory. Note that `node121728` has 2 links to `Sensor0021`, which means that if `node121728` is ON, then `Sensor0021` will be counted as ON twice (i.e., with a value of 2).

The reason for this double link is to address the problem of conflicting inheritance, where a node's parents disagree on the value for a feature. For example, in "The Nixon Diamond", we know that Republicans are War Hawks, Quakers are *not* War Hawks, and Richard Nixon is both a Quaker and a Republican. The question is then whether Nixon is a War Hawk. Figure 3.1 shows how this is decided. Essentially,

we add multiple links to the "War Hawk" node and count whether more inhibitory or excitatory links are active.



(a) War Hawk is UNSPECIFIED (b) War Hawk is ON (c) War Hawk is OFF

FIG. 3.1. **The Nixon Diamond.** Inhibitory links are shown as arrows, and excitatory links are those with small circles at the end. By counting the number of active links, we determine whether "War Hawk" is ON or OFF.

We allow that a feature may be missing, so we distinguish between absence of evidence and evidence of absence. This allows us to "fill in" the missing data using top-down reasoning, in a manner similar to (Hawkins & Blakeslee 2004).

3.3 An Energy Function for Ontologies and Parses

We provide an information theoretic function for evaluating ontology construction and parses using this ontology (Shannon 1948). In this sense, a "parse" is a small set of truths from which the other truths in an ontology may be derived.

Theorem 1. *Given a set of data D, and assuming a prediction-length prior on our model length, if ontology Ω minimizes the energy function $E(\Omega)$ in Equation 3.1, then*

this ontology maximizes the probability of the model given the data. This equation is

$$E(\Omega) = k|\Omega| - \sum_i \max_{R_i} \left(\log_2 \left(P(D_i|R_i, \Omega) \right) + k \sum_{r \in R_i} \log_2 P(r|\Omega) \right) \tag{3.1}$$

Also, given a fixed ontology Ω and datum D_i, the most probable set of parses R_i, are those that minimizes $E_R(R_i)$, where

$$E_R(R_i) = -\log_2 \left(P(D_i|R_i, \Omega) \right) - k \sum_{r \in R_i} \log_2 P(r|\Omega) \tag{3.2}$$

Proof. Given a data set D, where D_i is a set of binary features, we want to find the most probable ontology Ω and set of parses R. That is, we want to find Ω and R to maximize maximize $P(ontology = \Omega, parse = R|data = D)$. Here R_i is the parse for datum D_i, meaning that R_i is a set of binary features, which when stated should give a high probability to D_i. In this derivation, we'll use the convention that $P(\Omega)$ is shorthand for $P(ontology = \Omega)$, and similar shorthand for $P(D)$ for $P(data = D)$, etc.. Using Bayes's rule, we get that $P(\Omega, R|D)$ is proportional to $P(D|\Omega, R)\, P(\Omega, R)$. Since each datum $D_i \in D$ is parsed independently (by R_i), we get

$$P(D|\Omega, R) = \prod_i P(D_i|R_i, \Omega)$$

Substituting this back in, and exploiting the fact that all the R_is are independent

of each other, we want to find Ω and R to maximize

$$\begin{aligned} P(\Omega, R|D) &\propto P(\Omega, R) \prod_i P(D_i|R_i, \Omega) \\ &= P(\Omega) P(R|\Omega) \prod_i P(D_i|R_i, \Omega) \\ &= P(\Omega) \prod_i P(R_i|\Omega) P(D_i|R_i, \Omega) \end{aligned}$$

Given Ω and D_i, we can independently find a R_i, so if we just want to maximize Ω, we use

$$P(\Omega|D) \propto P(\Omega) \prod_i \max_{R_i} (P(R_i|\Omega) (P(D_i|R_i, \Omega)))$$

Following Wolff (Wolff 2003), we assume a description length prior on Ω and R. Furthermore, since R_i is encoded in terms of Ω, we can use a Huffman-like encoding for the individual elements of R_i.

$$\begin{aligned} DL(R_i|\Omega) &= \sum_{r \in R_i} DL(r|\Omega) \\ &= \sum_{r \in R_i} -k \log_2 P(r|\Omega) \end{aligned}$$

Where k is a constant (usually 1) that converts between bits and probabilities. Likewise, the description length (DL) of Ω is given by $k|\Omega|$. $P(r)$ can be approximated by its frequency in Ω, since part of the ontology is the frequency of each of its elements.

Substituting these description length priors for $P(\Omega)$ and $P(R_i|\Omega)$

$$P(\Omega|D) \propto 2^{-k|\Omega|} \prod_i \max_{R_i} \left(P(D_i|R_i, \Omega) 2^{k \sum_{r \in R_i} \log_2 P(r|\Omega)} \right)$$

Taking $\log_2$ we yield

$$-k\left|\Omega\right| + \sum_i \max_{R_i} \left(\log_2 \left(P\left(D_i | R_i, \Omega\right)\right) + k \sum_{r \in R_i} \log_2 P\left(r|\Omega\right) \right)$$

Conversely, given data D, we want to find Ω to minimize $E\left(\Omega\right)$, where

$$E\left(\Omega\right) = k\left|\Omega\right| - \sum_i \max_{R_i} \left(\log_2 \left(P\left(D_i | R_i, \Omega\right)\right) + k \sum_{r \in R_i} \log_2 P\left(r|\Omega\right) \right)$$

which is Equation 3.1.

It follows that, given a fixed ontology Ω and datum D_i, we want to find the set of parses R_i that minimizes $E_R\left(R_i\right)$, where

$$E_R\left(R_i\right) = -\log_2 \left(P\left(D_i | R_i, \Omega\right)\right) - k \sum_{r \in R_i} \log_2 P\left(r|\Omega\right)$$

□

3.3.1 Probability Calculation for Discrete Evidence

Equation 3.1 requires that we're able to calculate the probability of a node M given the values of its surrounding nodes as evidence E (as *True* or *False*). An approximation is given by Theorem 2.

Theorem 2. *If we're given a node M in the ontology with evidence E, then an*

approximation to the probability that M is true is given by

$$P(M|E) \approx \frac{(C(M)+1)\prod_{x\in E}\frac{C(x,M)+1}{C(M)+2}}{\left((C(M)+1)\prod_{x\in E}\frac{C(x,M)+1}{C(M)+2}\right)+\left((T-C(M)+1)\prod_{x\in E}\frac{C(x)-C(x,M)+1}{T+2}\right)}$$

where T is the total number of seen vectors, C is a function that returns the counts of occurrences, such that $C(x)$ is the number of times x occurred, and $C(x,M)$ is the number of times x and M occur together.

Proof. Applying Bayes's rule, we get

$$\begin{aligned} P(M|E) &= \frac{P(M)P(E|M)}{P(E)} \\ &= \frac{P(M)P(E|M)}{P(M)P(E|M)+P(\sim M)P(E|\sim M)} \end{aligned}$$

If we assume a significant amount of independence among the evidence in E, then this quantity becomes

$$P(M|E) \approx \frac{P(M)\prod_{x\in E}P(x|M)}{P(M)\prod_{x\in E}P(x|M)+P(\sim M)\prod_{x\in E}P(x|\sim M)}$$

If we assume a uniform prior on all our probabilities, then we can approximate these probabilities using the counts by using the *expected value* of a β distribution. Thus, for example, $P(x|M)$ is approximated by $\frac{C(x,M)+1}{C(M)+2}$. Substituting these quantities in the previous equation, we get

$$P(M|E) \approx \frac{\frac{(C(M)+1)}{T+2}\prod_{x\in E}\frac{C(x,M)+1}{C(M)+2}}{\left(\frac{(C(M)+1)}{T+2}\prod_{x\in E}\frac{C(x,M)+1}{C(M)+2}\right)+\left(\frac{(T-C(M)+1)}{T+2}\prod_{x\in E}\frac{C(x)-C(x,M)+1}{T+2}\right)}$$
$$= \frac{(C(M)+1)\prod_{x\in E}\frac{C(x,M)+1}{C(M)+2}}{\left((C(M)+1)\prod_{x\in E}\frac{C(x,M)+1}{C(M)+2}\right)+\left((T-C(M)+1)\prod_{x\in E}\frac{C(x)-C(x,M)+1}{T+2}\right)}$$

□

3.3.2 Probability Calculation Given Probabilities of Surrounding Evidence

In the case where we don't know the actual value of the surrounding evidence for node M, but instead are given V, which is the probability estimates for the surrounding evidence, we can estimate the probability of M given V using Theorem 3

Theorem 3. *If V is the surrounding evidence (such that $V(x)$ is our probability estimate for x), and we assume independence among all M's surroundings, we get:*

$$P(M|V) = P(M)\prod_{x\in E}\left(\frac{P(x|M)V(x)}{P(x)}+\frac{(1-P(x|M))(1-V(x))}{1-P(x)}\right)$$

Proof. If we consider each of the $2^{|E|}$ combinations of True/False for the elements of

E, then we can expand our probability for $M|V$ to be

$$\begin{aligned}
P(M|V) = {} & P(M|x_1, x_2, x_3, \cdots, x_n)\, P(x_1, x_2, x_3, \cdots, x_n|V) \\
& + P(M|\bar{x_1}, x_2, x_3, \cdots, x_n)\, P(\bar{x_1}, x_2, x_3, \cdots, x_n|V) \\
& + P(M|x_1, \bar{x_2}, x_3, \cdots, x_n)\, P(x_1, \bar{x_2}, x_3, \cdots, x_n|V) \\
& + P(M|\bar{x_1}, \bar{x_2}, x_3, \cdots, x_n)\, P(\bar{x_1}, \bar{x_2}, x_3, \cdots, x_n|V) \\
& + P(M|x_1, x_2, \bar{x_3}, \cdots, x_n)\, P(x_1, x_2, \bar{x_3}, \cdots, x_n|V) \\
& \cdots \\
& + P(M|\bar{x_1}, \bar{x_2}, \bar{x_3}, \cdots, \bar{x_n})\, P(\bar{x_1}, \bar{x_2}, \bar{x_3}, \cdots, \bar{x_n}|V)
\end{aligned}$$

Where

$$\begin{aligned}
& P(M|x_1, x_2, x_3, \cdots, x_n)\, P(x_1, x_2, x_3, \cdots, x_n|V) \\
& = P(M|x_1, x_2, x_3, \cdots, x_n)\, V(x_1)\, V(x_2)\, V(x_3) \cdots V(x_n) \\
& = \frac{P(x_1, x_2, x_3, \cdots, x_n|M)\, P(M)}{P(x_1, x_2, x_3, \cdots, x_n)} V(x_1)\, V(x_2)\, V(x_3) \cdots V(x_n) \\
& = P(M)\, \frac{P(x_1|M)\, P(x_2|M)\, P(x_3|M) \cdots P(x_n|M)}{P(x_1)\, P(x_2)\, P(x_3) \cdots P(x_n)} V(x_1)\, V(x_2)\, V(x_3) \cdots V(x_n) \\
& = P(M)\, \frac{P(x_1|M)\, V(x_1)}{P(x_1)} \frac{P(x_2|M)\, V(x_2)}{P(x_2)} \frac{P(x_3|M)\, V(x_3)}{P(x_3)} \cdots \frac{P(x_n|M)\, V(x_n)}{P(x_n)}
\end{aligned}$$

So our equation becomes:

$$
\begin{aligned}
P(M|V) = P(M)\,(& \\
& \frac{P(x_1|M)\,V(x_1)}{P(x_1)}\frac{P(x_2|M)\,V(x_2)}{P(x_2)}\frac{P(x_3|M)\,V(x_3)}{P(x_3)}\cdots\frac{P(x_n|M)\,V(x_n)}{P(x_n)} \\
+ & \frac{P(\bar{x_1}|M)\,V(\bar{x_1})}{P(\bar{x_1})}\frac{P(x_2|M)\,V(x_2)}{P(x_2)}\frac{P(x_3|M)\,V(x_3)}{P(x_3)}\cdots\frac{P(x_n|M)\,V(x_n)}{P(x_n)} \\
+ & \frac{P(x_1|M)\,V(x_1)}{P(x_1)}\frac{P(\bar{x_2}|M)\,V(\bar{x_2})}{P(\bar{x_2})}\frac{P(x_3|M)\,V(x_3)}{P(x_3)}\cdots\frac{P(x_n|M)\,V(x_n)}{P(x_n)} \\
+ & \frac{P(\bar{x_1}|M)\,V(\bar{x_1})}{P(\bar{x_1})}\frac{P(\bar{x_2}|M)\,V(\bar{x_2})}{P(\bar{x_2})}\frac{P(x_3|M)\,V(x_3)}{P(x_3)}\cdots\frac{P(x_n|M)\,V(x_n)}{P(x_n)} \\
+ & \frac{P(x_1|M)\,V(x_1)}{P(x_1)}\frac{P(x_2|M)\,V(x_2)}{P(x_2)}\frac{P(\bar{x_3}|M)\,V(\bar{x_3})}{P(\bar{x_3})}\cdots\frac{P(x_n|M)\,V(x_n)}{P(x_n)} \\
\cdots & \\
+ & \frac{P(\bar{x_1}|M)\,V(\bar{x_1})}{P(\bar{x_1})}\frac{P(\bar{x_2}|M)\,V(\bar{x_2})}{P(\bar{x_2})}\frac{P(\bar{x_3}|M)\,V(\bar{x_3})}{P(\bar{x_3})}\cdots\frac{P(\bar{x_n}|M)\,V(\bar{x_n})}{P(\bar{x_n})} \\
) &
\end{aligned}
$$

Which reduces to:

$$P(M|V) = P(M)\prod_{i=1}^{n}\left(\frac{P(x_i|M)\,V(x_i)}{P(x_i)} + \frac{P(\bar{x_i}|M)\,V(\bar{x_i})}{P(\bar{x_i})}\right)$$

□

3.4 Parsing with An Ontology

Our parsing algorithm starts with an ontology and an initial truth setting. Regardless of where the nodes are in the ontology, the nodes that are *set* to be true or false are called raw sensor nodes. Parsing focuses on "explaining" the raw sensor

nodes by stating truth values for other nodes.

Additionally, each node's truth value is dependent on the values for its neighbors. Each node is either ON, OFF, or UNSPECIFIED. Additionally, each node that isn't UNSPECIFIED is set to one of the values of either "EXPLAINED" or "FORCED". If a node is set to EXPLAINED, this means that its value is derivable from other nodes in the ontology. FORCED means not EXPLAINED, as if the nodes value has been set to be ON or OFF by a force external to the ontology. In principle, all the truth settings for the nodes in an ontology can be derived from the set of nodes that are FORCED. With this approach, we don't need to cover the full Markov blanket for a node. That is, we assume approximate independence among the parents of a node, then we need only know that a node n is EXPLAINED (i.e., *some* parent p is ON and $P(n|P) \approx 1$), but not *which* parent explains it. This is particularly useful if a node has hundreds of parents.

Figure 3.2 shows an example parse and truth-derivation. Although all nodes except node 11 are ON, the "parse" is only nodes 16 and 10, with the specification that the OR node 16 is instantiated by node 14. For us, a parse is a small set of truths from which the other truths in a network may be derived. The set of nodes 1 - 9 would also suffice as a parse, but is substantially larger, and therefore less preferred.

3.4.1 Optimal Parsing is NP-Hard

Here we prove that optimal parsing in NP-Hard, because The Set Covering Problem is reducible to The Parsing Problem. A parse R_i for an ontology Ω is a set of

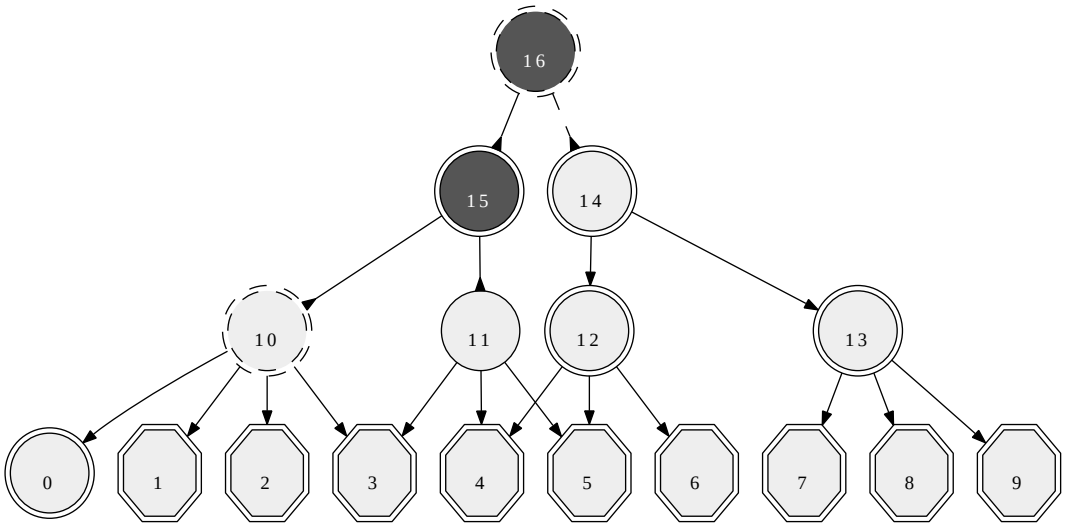


FIG. 3.2. **An example parse and truth-assignment.** Light nodes are ANDs and dark nodes are ORs. Nodes with double boundaries are ON and the other nodes are OFF or UNSPECIFIED (in which case they go to their default value, OFF). Raw data nodes are shown as octagons. Nodes with dotted borders are nodes "in" the parse. Here, nodes 1 - 9 are set to ON, which causes all nodes but node 11 to turn ON. The "parse" for this situation is nodes 10 and 16 and the specification that node 16 is instantiated by node 14 (at a cost of 1 bit), which is represented by the dashed line from node 16 to node 14.

propositions, such that each proposition $r \in R_i$ is either `true` or `false`. Every proposition $r \in R_i$ corresponds to a node in the ontology Ω (setting that node to `true` or `false`), but nodes in Ω may be unspecified.

Given an ontology Ω, and a feature-set D_i, we want to find the parse R_i that maximizes

$$\log_2 \left(P \left(D_i | R_i, \Omega \right) \right) + k \sum_{r \in R_i} \log_2 P \left(r | \Omega \right) \tag{3.3}$$

Note that $\log_2 P(x) \leq 0$, so we're really minimizing the magnitude of this function.

Theorem 4. *Optimal parsing –finding R_i to maximize the expression in Equation 3.3– is NP-hard.*

Proof. **The Set Covering Problem**

The Set Covering Problem is one Karp's 21 original NP-complete problems (Karp 1972). Given a set of elements $\mathcal{U}$, and a family $\mathcal{S}$ of subsets of $\mathcal{U}$ (where we assume that every element in $\mathcal{U}$ occurs in at least one set in $\mathcal{S}$), a covering set $\mathcal{M}$ is a subset of $\mathcal{S}$ such that every element of $\mathcal{U}$ is contained in at least one of the sets in $\mathcal{M}$. If this is the case, we'll say that $\mathcal{M}$ *covers* $\mathcal{S}$. The decision problem is whether it's possible to construct a covering set such that $|\mathcal{M}| < k$ for some integer k.

The Set Covering Problem is reducible to The Parsing Problem

Given a particular set covering problem $\mathcal{S}$ and k, we can construct an equivalent ontology Ω and feature-set D_i such that it's possible to cover $\mathcal{S}$ using fewer than k subsets if and only if it's possible to parse D_i at a cost of higher than $-k$.

To do this construction, for every element $u \in \mathcal{U}$ we create a corresponding node

u' in Ω, and also include this node in D_i and set its value to `true`. (These correspond to the input nodes in Ω.) Then, for every set $s \in \mathcal{S}$, we add a new node n to Ω and set the probabilities such that for every element $s_i \in s$, we find its corresponding node u' in Ω. We set the probabilities such that $P(u'|n) = 1$ and $P(u') = 0$ if none of its parents are `true`. Finally, we set $P(n) = .5$ (so it takes 1 bit to specify the truth value for these). Since the probability of all children of a node being `true` when that node is `true`, and the cost of specifying that a node is `true` is 1, our cost is simply the number of nodes we need to specify as `true`. Thus, our parsing task becomes the task of finding an optimal parse in this construction.

One difference between the parsing task and the set covering task is that, in the parsing task, we're allowed to specify that individual data items are true or false (though, in the minimal set covering problem, we're not allowed to include individual elements from $\mathcal{U}$ in $\mathcal{M}$). However, allowing individual elements won't help reduce our cost in either case because for any element u, we can (at the same price) include a set in $\mathcal{M}$ or a parent of u' that already includes u or u'. □

3.4.2 Our Energy Function Yields Sensible Parses

As a sanity test, I ran the following algorithm on an ontology Ω. I constructed all $3^{|\Omega|}$ possible parses, by choosing all combinations of nodes in Ω where each node was set to either `True`, `False`, or `Unspecified` (i.e., not included in the parse). For each potential parse R', I evaluated $E(R')$ by converting the network into a full Bayesian network and doing exact Bayesian inference (using an off-the-shelf Bayesian inference engine) to compute the probabilities $P(D_i|R')$.

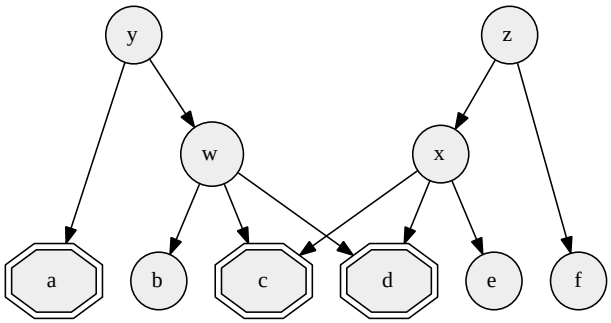


FIG. 3.3. **An example AND network.** Nodes a, c, and d are all set as raw data nodes to ON. The best parse for this situation is y being set to ON.

For the ontology shown in Figure 3.3, our evidence includes a, c, and d (all set to `True`)

The algorithm described above correctly concluded the best parse to be $\{y\}$. Below are the scores for the 10 top scoring parses:

Parse	**Score**	**Parse**	**Score**
$\{y\}$	0.3750	$\{a, w\}$	0.2146
$\{y, x\}$	0.2187	$\{c, y\}$	0.2127
$\{a, d\}$	0.2180	$\{w\}$	0.2127
$\{a, x\}$	0.2162	$\{y, z\}$	0.2115
$\{a\}$	0.2146	$\{a, c\}$	0.2099

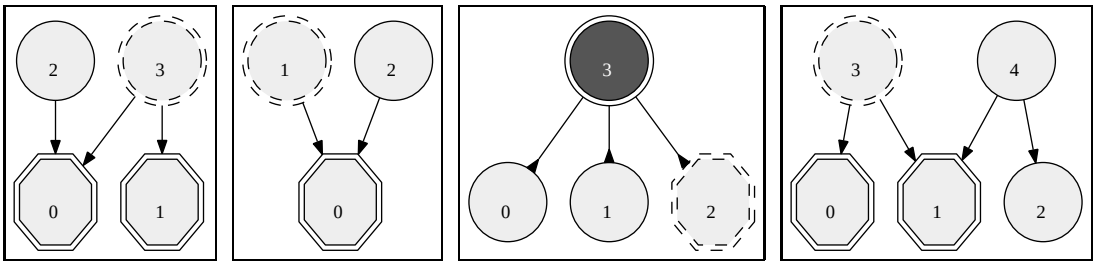
Though our energy function may be sensible, this technique is not. Searching the power set of the items in Ω is exponential in $|\Omega|$. Furthermore, exact inference in Bayesian networks is NP-hard (Cooper 1990). Assuming $P \neq NP$, this yields a runtime of at least $O\left(2^{n^2}\right)$ for the above described algorithm. Therefore, more pragmatic algorithms will be necessary, especially if we plan on working with ontologies

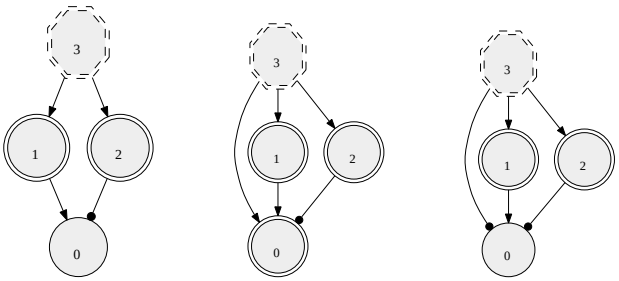
that have millions of nodes. We can exploit the lattice structure of the ontology, and we can accept approximations both in inferences and in parses.

The following graphs are test cases. For each of these, I set the values for the "raw inputs" (shown as octagonal nodes) and exhaustively searched over all the truth setting combinations for the remaining nodes. The results are shown here. Nodes that are ON have a double border. For each of these, the results match with what makes intuitive sense. For example, networks nixonn, nixonnP, and nixonnW are our Nixon Diamond examples from Section 3.2. In nixonn, node 0 (the "War Hawk" node) is at its default value False. In nixonnP the node is set to False, and in nixonnW, it's ON.

We did an exhaustive search over all the truth settings for a number of small networks. In each case, the algorithm agreed with what we wanted the truth setting to be. For example, we ran the "Nixon Diamond" example from Figure 3.1. In Figure 3.4, the top node "Nixon" is set as ON. A node is displayed as ON if it has a double border. A node is shown as FORCED if its border is dotted. An octagonal node is an input node (where the value has been specified externally to be ON or OFF).

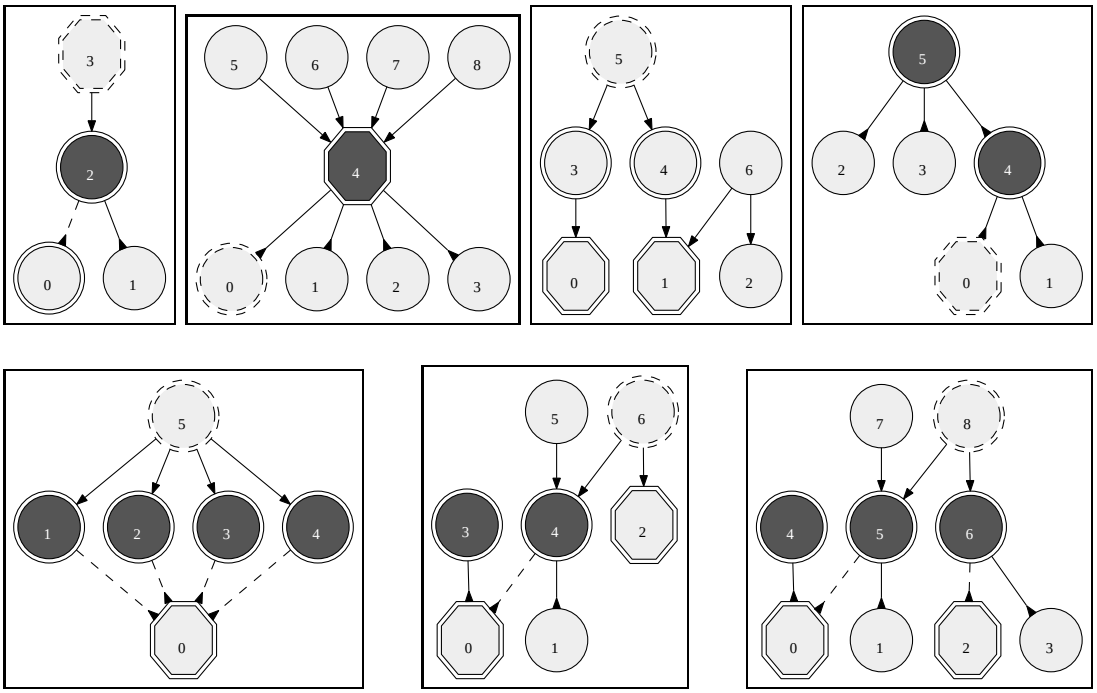
A sampling of the networks we tested is below.

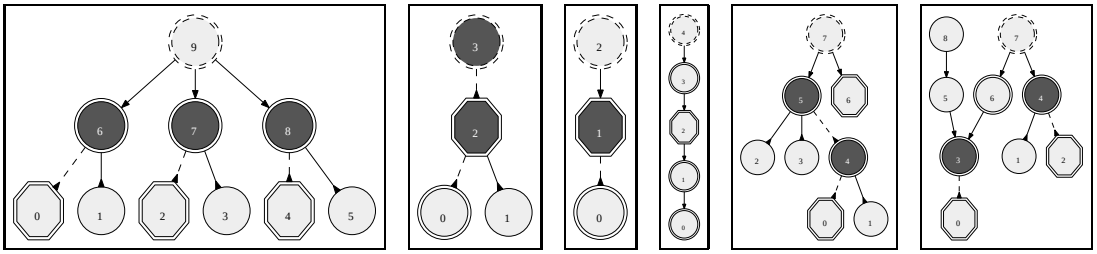




(a) War Hawk is UNSPECIFIED (b) War Hawk is ON (c) War Hawk is OFF

FIG. 3.4. **How our representation handles The Nixon Diamond.** Inhibitory links are shown as arrows, and excitatory links are those with small circles at the end. By counting the number of active links, we determine whether "War Hawk" is ON or OFF.





3.4.3 The Parsing Algorithm

Given an ontology Ω and a set of "input" nodes n for which we're told the truth settings (usually raw sensor nodes), we find a truth setting by hillclimbing (or "hill-descending", technically) on Equation 3.2. That is, we randomly initialize each non-input node to ON or OFF, then we switch these values and hillclimb by choosing the configuration that minimizes Equation 3.2. After a specified number of random restarts, we return the configuration that produced the lowest value for Equation 3.2.

Given a set of truth-settings, we find the "parse" (a subset of the truth-settings from which the truth-settings for all the remaining nodes can be derived) by evaluating whether each node is "FORCED" or "EXPLAINED". This is done locally. For each node, we compute whether the node's parents cumulatively want the node to be ON or OFF (by counting the number of excitatory minus inhibitory links from the ON parents). If a node is ON, but its parents wish it to be OFF (i.e., it has a non-positive number of excitatory links), then that node is FORCED. Likewise, if a node's parents wish it to be ON and the node is set to OFF, then that node is also FORCED. Otherwise the node is EXPLAINED. Our parse is the set of nodes that are FORCED.

Given enough restarts, this algorithm will find the optimal parse (i.e., the parse

that minimizes our energy function) because it's possible (though very unlikely) that a restart will begin with the optimal parse. Since there are 2 truth settings (ON and OFF) for each node, this yields 2^n possible parses, where n is the number of nodes. At least one of these parses minimizes our energy function, so on average, we'd expect to the optimal parse (with no hillclimbing) in $O(2^n)$ restarts.

To evaluate this algorithm, we ran our parsing algorithm on the test networks from Subsection 3.4.2. For each of these, we got the same parses that we get by doing exhaustive search. This parsing algorithm is part of the semi-supervised learning algorithm described in Section 3.6, which we use to evaluate the utility of this algorithm as part of a larger system.

3.5 Building An Ontology

To start, the ontology building system uses the energy function in Equation 3.1 to decide what ontology is best for a data set. The energy function is based on principles of probability theory and information theory (Shannon 1948), but basically can be thought of as a variant of Ockham's Razor, where we try to find the ontology that helps us express the input data using the fewest bits.

AND nodes are formed by chunking, which creates nodes for large and frequently occurring feature-sets. For example, if we see the set A, B, C, D often, then we can create a new node M that is shorthand for this sequence. We can use M to compress the set A, B, C, D, E, F down to the set M, E, F.

OR nodes are formed by merging. If 2 or more nodes are *interchangeable* with

respect to their predictive power for other features, then we *merge* these nodes, thus forming an equivalence class. A new OR node is created which represents the disjunction of the nodes in this set. For example, if we often see A, C, ... and B, C, ... (but rarely A, B, C, ...), then we might create a new node N which is true if A OR B are true. Then we can represent the set A, C and B, C both as N, C, which we can then chunk (since there are 2 of them) as L. So the set A, C, E gets represented as L, E. Note that merging decreases the accuracy of the ontology, but allows for extra chunks that reduce the ontology's description length. Merging also helps generalize concepts.

In searching for both ANDs and ORs, we exploit the property that patterns that are useful for compression tend to be those that are frequent. If a pattern is found n times, then these occurrences are n places where this pattern can be used to characterize the data. This principle —that the most useful patterns are the most frequent— works in our favor because patterns that are frequent are also easier to find.

3.5.1 Chunking

The Cruncher, first described in (Pickett & Oates 2005), is a simple representation framework and algorithm based on minimum description length for automatically forming an ontology of concepts from attribute-value data sets. Although unsupervised, when The Cruncher is applied to the UCI Zoo Database (Blake & Merz 1998), it produces a nearly zoologically accurate categorization. Like PolicyBlocks, The Cruncher finds useful macro-actions in Reinforcement Learning. The Cruncher can also learn models from uninterpreted sensor data. In the same paper, we also discuss

advantages The Cruncher has over concept lattices and hierarchical clustering.

Given a collection of sets of attribute-value pairs, where each attribute's value is from a finite alphabet, The Cruncher produces a concept ontology which uses inheritance to compress the collection of attribute sets. One can "flatten" this ontology by computing the inheritance of every node in it, and this flattened ontology will contain the original collection of attribute sets. The Cruncher uses a greedy approach for reducing the description length of the ontology (which is initially just the collection of attribute sets). The description length is defined as the number of links in the ontology, be they "is-a" or "has-a" links, where an "is-a" link designates that one node inherits from another, and a "has-a" link specifies an attribute that a node has and that attribute's value. (Whether the number of nodes was also included in the description length did not significantly affect our results partly because this number usually closely corresponds with the number of links.) The Cruncher generates candidate concepts by finding the "intersections" of subsets of the current items in the ontology. These candidates are then evaluated by determining the reduction in description length if each were to be adopted as concepts in the ontology. If no candidate reduces the description length, then The Cruncher halts. If a candidate is selected, then it is added to the ontology, and all other concepts in the ontology inherit from it if it reduces their description length. If there is a contradiction in the value assigned to an attribute by the nodes from which a concept inherits, that term is simply discarded. Furthermore, if a concept has an attribute, but the node from which it inherits has a different value for that attribute, then the concept states what its value is for that attribute. Thus, exceptions are allowed. See Table 3.1 for details of the algorithm.

Algorithm The basic Cruncher algorithm is shown in Table 3.1. Essentially, this searches for intersections among existing nodes and proposes these as candidates for new concepts. Each candidate is evaluated by how much it would compress the ontology, then the best candidate is selected and added to the nodes, and the process is repeated until no candidates are found that further reduce the description length of the ontology.

The runtime of this algorithm depends on whether one generates all possible candidate concepts, which, theoretically, can be exponential in the number of sets of original attribute pairs. In practice, one can successfully generate ontologies by randomly generating only a subset of these candidates, thus yielding a polynomial time algorithm. There is also an incremental version of this algorithm which works by inserting one new concept at a time, and generating candidates by intersecting that node with each of the other concepts in the ontology. The top candidate is selected (or none if no candidate yields a decrease in description length), then this candidate is inserted into the ontology following the same procedure.

In practice, we needed to randomize the mergeAll function because to search all candidates exhaustively for large datasets required too much time and memory. Instead, we randomly chose nodes, then computed the intersections of these nodes' children. We call this algorithm "basicCrunch".

We tested several modifications of the basicCrunch algorithm. One variant, which we call "salienceCrunch", deals with the idea of "salience" or "moderate novelty". When choosing nodes from which to find intersections, a simple search strategy instead of uniform random selection would be to decrease the probability of selecting a node in proportion to the number of times it has already been selected. Inversely,

Table 3.1. **The Cruncher algorithm.**

```
// Returns an ontology that compactly expresses S
Cruncher(S) (where S is a set of attribute-value sets)
  let B be a set of ConceptNodes such that S
    foreach attribute-value A in S there is a corresponding ConceptNode c in B
      such that A ∈ c.hasA and c.isA = ∅.
  while we are still decreasing the description length of B
    candidates = mergeAll(B)
    compute score(B, candidate) foreach element in candidates
      let best be the highest scoring candidate
      if score(B, best) > 0 then let B = replaceBest(B, best) + best
  return B

// Returns all possible intersections of subsets of B
mergeAll(B)
  let Closed and New be new empty sets of attribute-value sets
  let Open = B
  while |Open| ≥ 0
    foreach pair of attribute-value sets F1, F2 ∈ Open
      if F1 ∩ F2 ∉ New add F1 ∩ F2 to New
    foreach F ∈ Open
      remove F from Open
      if F ∉ Closed add F to Closed
    foreach F ∈ New
      remove F from New
      if F ∉ Closed add F to Open
  return Closed

// The decrease in description if candidate is used as a concept
score(B, candidate) = Σ_{n∈B} max(0, nodeScore(n, candidate)) - |candidate.hasA|
replaceBest(B, best) foreach n ∈ B if nodeScore(n,best) > 0 then replace(n, best)

// Returns the decrease in description length if n were to inherit from candidate
nodeScore(n, best)
  return the decrease in the description length for n and n after replace(n, best)

// Make n inherit from best
replace(n, best)
  let flat be the flattened value of n
  foreach a ∈ n.hasA ∩ best.hasA remove a from n.hasA
  foreach a ∈ best.hasA such that ∃b ∈ n where attributes of a and b match, but the values disagree
    add a to n if it's not there already
  foreach a ∈ best.hasA but a ∉ flat
    add b to n where b's attribute is the same as a's but b's value = ignore
    (During flattening the ignore value causes the inherited attribute-value to be discarded.)
```

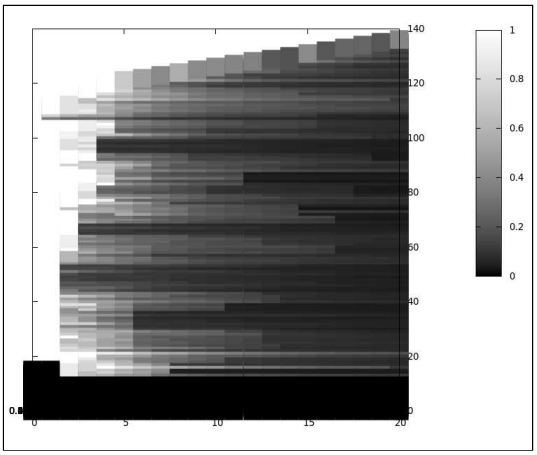


FIG. 3.5. **The "salience" mechanism in action.** The x axis is the iteration number (so at $x = 1$, we're creating the 1st new chunk). The y axis is the concept number, with the newest concepts at the top. The grey value is the salience score for each node at the start of a particular round. The 16 black values at the bottom are the sensor nodes, which have no children (and therefore, nothing to "explain"), so the salience for these is 0.

we want to increase this probability in proportion to the amount of description length we expect we might gain from crunching it. This way, we pay "attention" to new nodes or nodes we haven't looked into before, or nodes that might have a lot left to crunch away.

To implement this idea, each node had an integer p keeping count of the number of times it was picked, it also had an integer u representing the number of children it had (i.e., the number of "unexplained" children it had). Then a node's "salience value", directly proportional to the probability of it being chosen, was $\frac{u}{1+p}$. Figure 3.5 shows this mechanism in action during a build of an ontology. There are 2 ways a node can lose salience: it can be "explained" or it can lose salience because we've fruitlessly tried explaining it too many times.

Table 3.2. **Comparison of different crunching algorithms.** The Energy Score and Computational Cost are shown for 3 different crunching variations. The 95% confidence interval is shown for each value.

Algorithm	**Avg. Energy Score**	**Avg. Cost**
basicCrunch	2605.01±12.02	4208.27±134.25
salienceCrunch	2580.30±06.35	4701.34±114.58
parentCrunch	2528.06±10.31	4121.94±148.63

Another variant is called "parentCrunch", which constrains its search for intersection by picking a node, then finding the intersection of 2 of the node's parents. This intersection is guaranteed to be non-empty (because the node itself will be in it). This turned out to be a useful approach in practice.

A comparison of these 3 algorithms: basicCrunch, salienceCrunch, and parentCrunch is shown in Figure 3.6. Since these algorithms are randomized, we can vary the number of candidates for each round. The number of candidates evaluated is the x axis, and the y axis is the size of the resulting ontology for the UCI Zoo dataset. We want lots of compression, so a lower score is better. As shown in Table 3.2, parentCrunch did the best, having the best average amount of compression for a fixed amount of computation. We also tried variants of hillclimbing, where we evaluated modifications of promising candidate chunks, but the computation involved in these evaluations was usually not worth the gains compared to trying entirely new candidate chunks. We also tried variants of all these tricks, but nothing turned out to be significantly better than parentCrunch, so we use this as our crunching algorithm.

Experiments To test the general applicability of The Cruncher, we chose a diverse set of domains to which we applied our algorithm. For example, the UCI

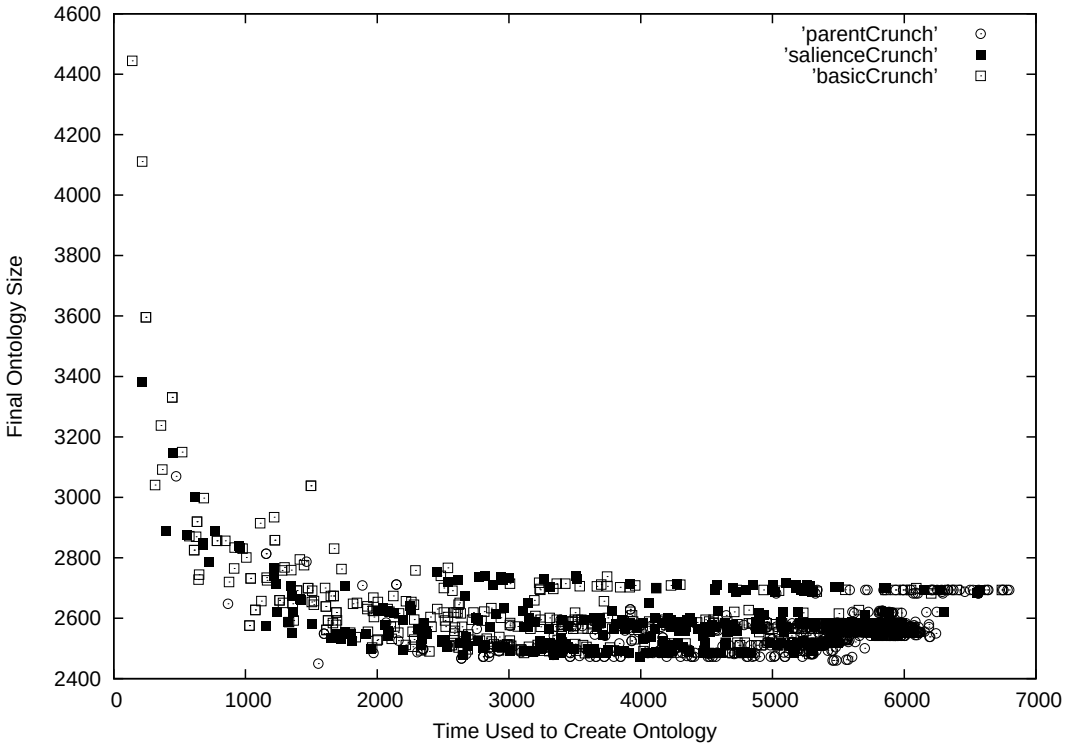


FIG. 3.6. **A comparison of the best 3 search strategies.** Each point is a complete run of the algorithm on the UCI Zoo dataset. The x axis is the total amount of computation time used (in terms of the number of candidate nodes evaluated), and the y axis is the description length of the resulting ontology. For this plot, Parent Crunch has the best computation to compression ratio, since it produced the best compressed ontology at a low computational cost.

Zoo Database is useful for gaining insight about The Cruncher's ontologies. We show the compression that can be achieved using The Cruncher on several UCI datasets (Blake & Merz 1998) compared to using a standard compression algorithm alone (we use Lempel-Ziv (Ziv & Lempel 1977)).

We also apply The Cruncher to reinforcement learning to create an algorithm called PolicyBlocks. This algorithm demonstrates the use of The Cruncher on domains other than classification or compression tasks. In these Reinforcement Learning domains, there is a definite performance measure (cumulative reward).

Compression Performance of The Cruncher We ran The Cruncher on several datasets from the UCI machine learning repository. For example, the ontology that was created by The Cruncher for the UCI "zoo" dataset is shown in Figure 3.7. An interesting outcome is that the animals are arranged according to their classification even though this classification data was never provided. For example, there is almost a one-to-one correspondence to the animals that inherit from the black node in the lower right and the class Mammal. (The Tuatara and the Tortoise are the only exceptions.) Birds, Fish, and Invertebrates are likewise grouped together. The utility of allowing exceptions is demonstrated by the Platypus, which is classified as an egg-laying mammal (even though mammals are asserted as not laying eggs). The utility of having multiple inheritance is demonstrated by the case of the Penguin, which shares traits with both aquatic life and birds. In Section 3.6, we show how Ontologies created from unlabeled data can be used to create a semi-supervised learning algorithm that's able to learn categories from a handful of positive training instances.

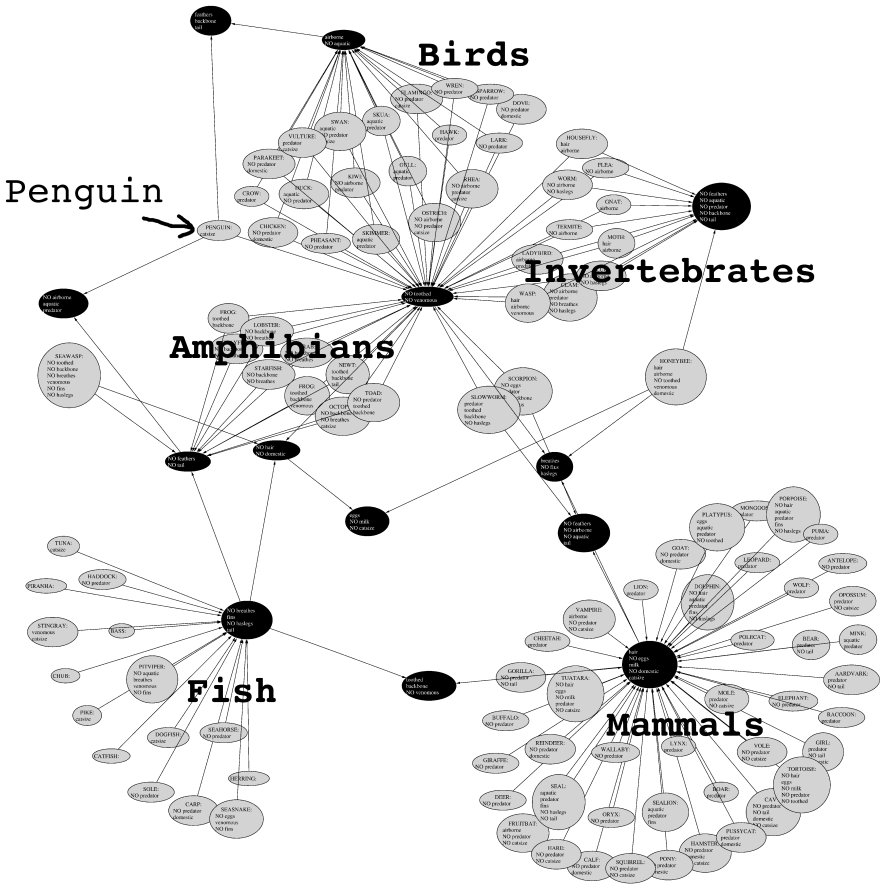


FIG. 3.7. **The automatically created zoo ontology.** Arrows represent inheritance, and the attributes and their values are listed in the nodes. Grey nodes are the original concepts in the database, and black nodes were created by The Cruncher. Mammals are grouped in the lower right, fish in the lower left, and the other three major clusters, from left to right, are, for the most part, reptiles/amphibians, birds, and invertebrates. Note, the multiply inheriting Penguin in the upper left. The class of birds is divided just for the Penguin, and the Penguin shares traits with the aquatic animals ("not airborne", "aquatic", "predator").

Table 3.3 shows the results of compressing the text files for various datasets using the Lempel-Ziv (LZ) compression algorithm (Ziv & Lempel 1977) alone compared to compressing the ontology generated by The Cruncher for the datasets. Since we're only chunking, both of these methods are lossless. For the LZ+Cruncher algorithm, we first crunched the dataset, then wrote the feature-sets of the crunched ontology to a text file, then compressed that file with LZ compression. The compression using The Cruncher was significantly better for every dataset we tested. For example, the "connect-4" dataset compressed to about quarter of its size that LZ alone could compress it to. This is because LZ compression is specific to text files, and must keep track of things like the sequence of the file. The Cruncher, on the other hand, is for feature-set data, and so can exploit properties that are specific to feature-sets (that feature-sets are unordered, for example). In this case, The Cruncher is able to recover the original text files, because the features in the original text file were sorted alphabetically. In the case where these features aren't sorted, if there are n features, we would need an extra $\frac{1}{8}\log_2(n!)$ bytes to specify the ordering. The number of features in our files never exceeded 100, so at most this would be $\frac{1}{8}\log_2(100!) < 66$ extra bytes per text file. Using The Cruncher alone has the disadvantage that the output is written to a text file, so The Cruncher doesn't exploit the properties for text compression, such as making use of non-printable characters or compressing the feature names themselves.

PolicyBlocks: Creating Useful Macro-actions In Pickett and Barto (Pickett & Barto 2002), it was demonstrated that for policies in a Markov Decision Process, certain concepts, which are effectively those created in the first level of abstraction in The Cruncher, can be used as useful macro-actions. These macro-

Table 3.3. **Compression using The Cruncher.** Various UCI datasets are shown being compressed using the Lempel-Ziv compression algorithm on the raw data set vs. the Lempel-Ziv algorithm compressing the ontology created by The Cruncher. In all cases, The Cruncher helped to further compress the data compared to just LZ alone. The size for each file is given along with the percentage of the original size.

Dataset	Uncompressed		Cruncher		LZ		LZ+Cruncher	
connect-4	1,066,126	100%	55,315	5.19%	46,525	4.36%	11,283	1.06%
house-votes-84	104,611	100%	13,935	13.32%	6,441	6.16%	2,907	2.78%
kr-vs-kp	604,073	100%	35,421	5.86%	31,495	5.21%	6,747	1.12%
mushroom	1,494,117	100%	47,589	3.19%	32,846	2.20%	10,004	0.67%
nursery	392,027	100%	38,570	9.84%	11,351	2.90%	7,257	1.85%
SPECT	95,363	100%	11,346	11.90%	6,590	6.91%	2,417	2.53%
tic-tac-toe	199,291	100%	35,035	17.58%	9,987	5.01%	6,855	3.44%
zoo	29,427	100%	3,125	10.62%	1,353	4.60%	736	2.50%

actions outperform hand-chosen macro-actions such as getting to the "doorways" of the rooms in a grid-world. We started with a 20 by 20 grid-world structure (see Figure 3.8), and produced full policies leading to each of 20 randomly selected goal states. These policies are represented as a collection of 400 attribute-values, where the attributes are each of the 400 states, and the values are one of *up*, *down*, *left*, and *right*.

To illustrate the behavior of PolicyBlocks, we ran it on an 18 by 18 grid-world task. We then ran PolicyBlocks on a larger hydroelectric reservoir task, which is an abstraction of the task described by (Little 1955). For each task, we created 20 sample problems by randomly setting the reward structure (details below). We used policy iteration (with deterministic tie breaking) on these 20 problems to obtain a set of 20 sample policies. The discount factor, γ, was set always to .9.

Given a set of sample solutions, we found options according to PolicyBlocks and the SKILLS algorithm from (Thrun & Schwartz 1995), as well as the Reuse option

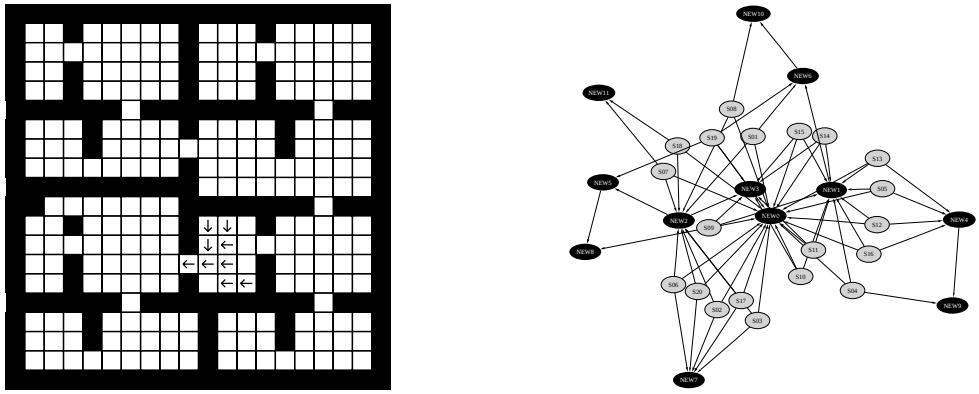


FIG. 3.8. **A macro-action ontology.** On the left is a macro-action generated by applying The Cruncher to a set of policies on a grid-world. The structure of the ontology is shown on the right. Each grey node corresponds to a full policy over the grid-world. Each black node is a sub-policy, or macro-action, that was produced by The Cruncher. For example, the macro-action encoded by the bottommost node in this ontology is that shown in the grid-world on the left. The arrows are "is-a" links, so every full policy can be thought of as the composition of all the sub-policies from which it inherits (in addition to the grey node's own modifications). Thus, each black node can be thought of as a "building block" for a full policy. (This is the origin of the term "PolicyBlocks").

from (Bernstein 1999). All option sets were augmented with the primitive actions. We compared the performance of Q-learning with these options sets with Q-learning using only primitive actions. For the grid-world, we also constructed a set of options by hand that we thought would be useful. These options had the goal of exiting a room by the north, south, east, or west faces (if such an exit existed). For the reservoir problem, it was not obvious to us what might be useful "hand chosen" options, so there is no comparison to these.

Using PolicyBlocks and SKILLS, we found the top 1, 2, 3, 4, and 5 options. Bernstein's Reuse algorithm allows for only one option, and has no parameters. In addition to number of options, SKILLS has the parameter η, which weights the relative importance of *description length* to *performance loss*. In general, a larger value

of η results in larger options. We set this parameter to .5, 1, and 2. For all of our comparisons, we set both the step size, α, and exploration rate, ϵ, to .01, .05, or .1. Initially each Q-value was set to 1. We evaluated the performance of these option sets by reporting the average accumulated reward over 100 new tasks for the grid-world and 10 new tasks for the reservoir domain.

Bernstein's Reuse option is a stochastic policy defined over all states (i.e., $I = S$). Given a set of solutions L to k Markov decision processes, this algorithm generates a set of k stochastic policies SP where $SP_i(a_j|s) = 1$ if $L_i(s) = a_j$ and $SP_i(a_j|s) = 0$ otherwise. The probability of taking action a in state s for the Reuse option is $\frac{1}{k}\sum_{i=1}^{k} SP_i(a, s)$.

SKILLS is more complicated. This algorithm seeks to minimize an energy function $E = PerformanceLoss + \eta DescriptionLength$ by gradient descent. Given a set of optimal Q-values to a set of k problems, $PerformanceLoss$ is the total decrease in value (from optimal for the sample problems) caused by constraining the action set to those defined by the option set for each state. That is, if there is an option defined for a state, not all primitive actions are necessarily available for that state. $DescriptionLength$ is the sum of the size of the options plus k for each state s for which no option is defined. To generate options, one must specify the size of the option set and η. For each option, SKILLS then modifies *usages* (i.e., how useful an option is for a particular sample problem), I, and π in order to minimize E.

The Grid-world Task

Our grid-world is an 18 by 18 grid (Figure 1) that is naturally divided into rooms with doorways. The grid-world has four stochastic actions: up, down, left,

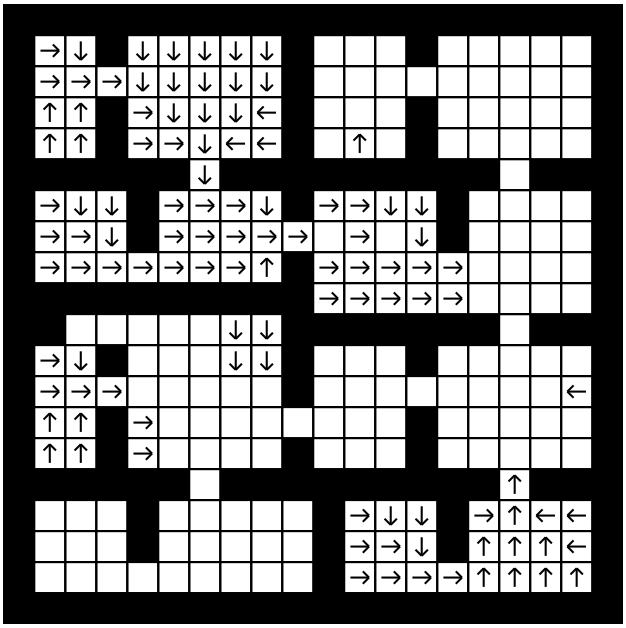


FIG. 3.9. **The grid-world with an example option.** Each black box is an obstacle, and each open box represents a state. The arrows indicate the actions for the subpolicy. Lack of an arrow for a box implies that the subpolicy is not defined for that state. Note that the option is discontiguous, and can be thought of as 8 separate contiguous options. This subpolicy was found in 15 of the 20 sample solutions, and has size 109 giving it a score of $109 \cdot 15 = 1,635$, making it the top option found. Other options found by PolicyBlocks followed this general pattern of taking the agent through 1 or 2 rooms to the door.

right, which each have a .9 probability of success. If an action does not succeed, the agent goes into one of the remaining legal directions each with equal probability. A direction is illegal if an obstacle blocks its path. Additionally, the agent has a deterministic "remain" action which allows the agent to stay in its current location. Entering a goal state yields a reward of 10, while all other transitions yield a reward of 0. While performing Q-learning, an agent is reset to a random location upon reaching the goal state. Grid-world sample problems were made by choosing a goal state randomly.

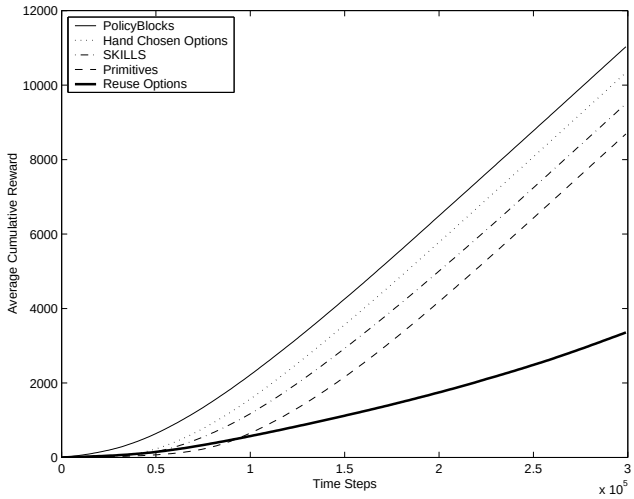


FIG. 3.10. **Option performance comparison for the grid-world.** The x-axis is time steps, and the y-axis is the cumulative reward averaged over 100 MDPs.

The Hydroelectric Reservoir Problem

A series of hydroelectric dams are positioned at distant points along a river. Each dam sells electricity to a different city, and each has a reservoir to collect water (Figure 3). Since there is no efficient electricity storage, the city uses and pays for the electricity as soon as it is generated. The price of the electricity is proportional to the city's demand. In addition, due to economic differences, different cities pay more or less for the same amount of electricity and for the same level of demand. Each dam has a reservoir, and an agent must decide which dams to close or open, thus producing electricity, letting the water flow down river, and lowering the level of the reservoir. The agent's goal is to maximize its total discounted profit. However, there is no "goal state" as in the grid-world. The first reservoir in the sequence gets a constant flow of water. This task is an abstraction of that described by Little (1955).

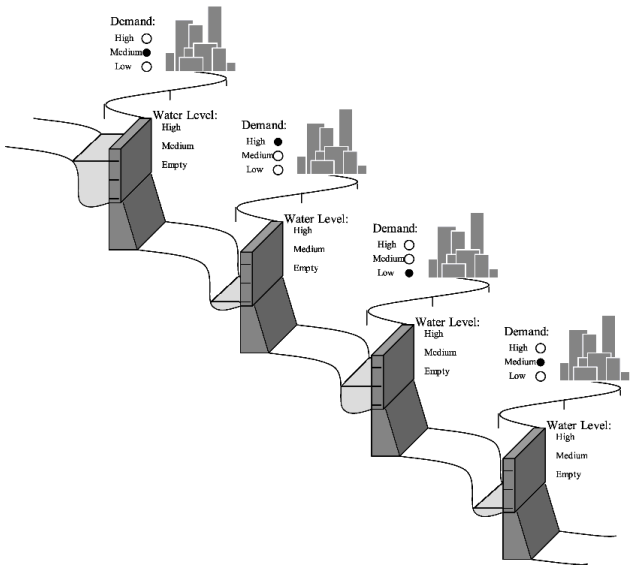


FIG. 3.11. **A sequence of hydroelectric reservoirs and dams.** Each of the 4 dams provides power for a single city. Each city's demand (and pay rate) varies with time. The agent must choose which dams to keep closed or open to produce electricity to sell immediately to its city. Opening a dam also sends some water to the next reservoir downstream. There is assumed to be a steady flow of water into the first reservoir.

In our specific example, there are 4 dams, each with a reservoir and a city. Each city i has a variable demand level D_i that can be either *high*, *medium*, or *low*, and a fixed pay rate $P_i \in (0, 1]$ that corresponds to the economic differences. Each dam i has a reservoir water level W_i of either *high*, *medium*, or *empty*. At each decision point, the agent can open or close each of the 4 dams, resulting in $2^4 = 16$ actions. If the agent opens a dam i, and it is not empty, a unit of electricity is produced, and the water level is decreased (from high to medium, or from medium to empty). The water flows down river, and the water level for the next reservoir W_{i+1} will be increased for the next decision point. The agent receives a pay from the city equal to its pay rate P_i if demand D_i is medium, twice the city's pay rate if demand is high,

and no pay at all if the demand is low. If the agent keeps the dam closed, nothing happens. Between decision points, the reservoirs fill (from empty to medium, or from medium to high) if the dam above them was opened (and water flowed). The top dam always fills. If a dam is already at a high level, and more water is coming from the dam above, the water is uselessly disposed of by flood prevention chutes, and the levels stay the same. The cities also change their demand levels stochastically: each has a .9 probability of remaining at the same level, and otherwise it transitions to an adjacent level (e.g., low to medium). The agent has a discount rate of .9 for each decision point. Since there are 3 levels for each dam, 3 levels for each the city's demand, and 4 cities and dams, there are $3^8 = 6,561$ different states.

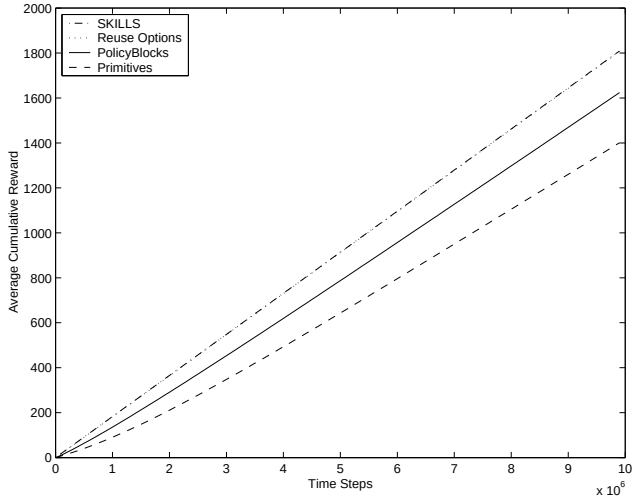


FIG. 3.12. **Option performance comparison for the reservoir problem.** The x-axis is time steps, and the y-axis is the cumulative reward averaged over 10 MDPs. The options generated by SKILLS faired best. A hair's breadth below (nearly atop SKILLS) is the Reuse option. PolicyBlocks performed better than using only primitives, but not as well as the other option selection schemes.

To apply PolicyBlocks, we consider instances of this problem where the difference

is the cities' pay rates P_i which are drawn according to a uniform distribution. Thus, the transition probabilities remain the same across problems, but the reward function varies yielding different optimal policies. For our experiment, we generated 20 sample problems where each P_i was chosen randomly uniformly from $(0, 1]$. We ran Q-learning on this problem for 10 new problems.

Results

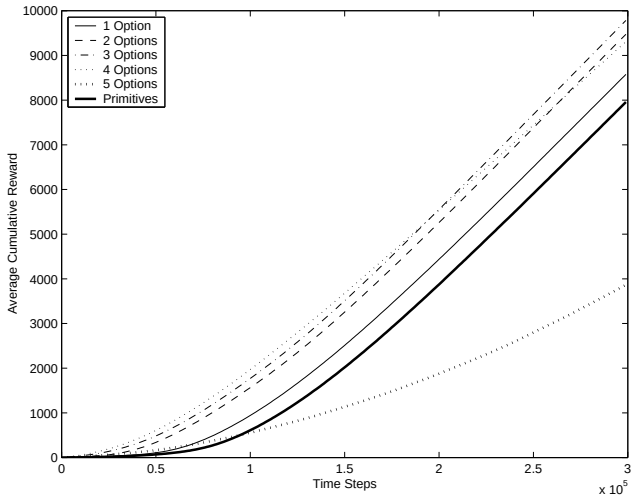


FIG. 3.13. **Effect of parameters for PolicyBlocks.** The x-axis is time steps, and the y-axis is the cumulative reward averaged over 100 grid-worlds. Using 2, 3, and 4 options has similar performance. Below these is using only the top option, and at bottom is the performance of using the top 5 options.

Figure 2 shows results for the grid-world comparison. Over all the parameter settings we tried, the best runs are shown for PolicyBlocks, SKILLS, the Reuse option, hand chosen options, and primitive actions alone. In all cases, option performance, in terms of cumulative reward was maximized among the parameters we tried when $\alpha = .1$ and $\epsilon = .01$. The optimal policy was found when the slope of the line becomes

constant. The top 4 options generated by PolicyBlocks outperformed even the "Hand Chosen Options". This was followed by SKILLS, using 2 options and setting $\eta = 1$. Although the Reuse option initially outperformed the use of primitive actions alone, this option failed to learn the optimal policy before 300,000 time steps.

Figure 4 shows results for the reservoir comparison. Again, we show the best runs for each algorithm. The options generated by SKILLS, using 5 options, setting $\alpha = .1$, $\epsilon = .01$, and $\eta = .5$, fared best. Just below that is the Reuse option. PolicyBlocks (setting $\alpha = .1$, $\epsilon = .05$, and using the top 3 options) performed better than using only primitives, but not as well as the other option selection schemes.

Figure 5 shows results for the effects of the number of options for PolicyBlocks on a grid-world. The performance is similar with 2, 3, or 4 options. Below these is the top single option, and at bottom is the performance of the top 5 options. All of these outperformed primitive actions except for the top 5 options, which failed to learn the optimal policy after 300,000 time-steps.

Since there are 2^{20} possible mergings, we were pleased to find that the vast majority of processor time was spent on Q-learning. Creating the options never took more than a few seconds once the sample solutions were found. This is because it was rare that more than 8 partial policies could be merged without having a null result.

3.5.2 Merging

As mentioned in Chapter 2, merging is the process of forming an equivalence class from items that appear in the same context. Equivalence classes can help reduce our energy function from Equation 3.1 because they allow us to form chunks where

we couldn't before. For example, suppose we have 1024 data items, where 256 are samples from each of the following 4 cases

$$\{E, F, \cdots, T\} \times \{e, f, \cdots, t\}$$
$$\{EE, FF, \cdots, TT\} \times \{ee, ff, \cdots, tt\}$$
$$\{E, F, \cdots, T\} \times \{ee, ff, \cdots, tt\}$$
$$\{EE, FF, \cdots, TT\} \times \{e, f, \cdots, t\}$$

For example, we have data items such as $\{F, m\}$, $\{GG, ii\}$, $\{H, ll\}$, and $\{GG, ii\}$. Suppose our ontology has about a million nodes, making the universal cost of a link roughly 20 bits. Then the initial cost of our 1024 items is $1024 \cdot 2 \cdot 20 = 40,960$ bits. If we form the relevant equivalence classes (e.g., $A = E|F|G \cdots |T$), this allows us to form *chunks* for each of our 4 cases. Then the cost of each of the 1024 items is the cost to call the chunk (20 bits) plus the cost of specifying the instantiation for each equivalence class. For example, the cost of specifying $F|A$ is $\log_2 |A| = \log_2 16 = 4$. So this cost is $2 \cdot 4 = 8$ bits. This brings the total cost for each pair using our equivalence classes to $8 + 20 = 28$ bits. Since there are 1024 pairs, this is $28 \cdot 1024 = 21,920$ bits. We also need to take into account the cost of constructing the chunks and the equivalence classes, which is 20 bits for each link, and there are 4 equivalence classes each with 16 links. Then there are 4 chunks, each with 2 links. This is a total of $4 \cdot 16 + 4 \cdot 2 = 72$ links, or $72 \cdot 20 = 1440$ bits. So the total new cost using equivalence classes is $21,920 + 1440 = 23,360$ bits. Thus, using equivalence classes, we save $17,600$ bits out of $40,960$, or we reduce our description length by 43%.

The question remains of how we find these equivalence classes to begin with. We do this by creating a new set of feature-sets where each feature-set is the *context* for a

Table 3.4. **An artificial grammar.**

$< S >$	::=	$\{< doubles > < M >\}$
$< M >$	::=	$< A >$ *(with probability .4)* $\mid < B >$ *(with probability .6)*
$< A >$	::=	$< 4 >, < 5 >, < 6 >, < 7 >$
$< B >$	::=	$< 1 >, < 2 >, < 3 >$
$< doubles >$	::=	$< AA > < BB > < CC > < DD > < EE > < FF > < GG >$
$< 1 >$	::=	1A | 1B | 1C | 1D | 1E | 1F
$< 2 >$	::=	2A | 2B | 2C | 2D
$< 3 >$	::=	3A | 3B | 3C | 3D | 3E | 3F
$< 4 >$	::=	4A | 4B | 4C
$< 5 >$	::=	5A | 5B | 5C | 5D | 5E
$< 6 >$	::=	6A | 6B | 6C | 6D | 6E | 6F
$< 7 >$	::=	7A | 7B
$< AA >$	::=	aa, | $\emptyset$
$< BB >$	::=	bb, | $\emptyset$
$< CC >$	::=	cc, | $\emptyset$
$< DD >$	::=	dd, | $\emptyset$
$< EE >$	::=	ee, | $\emptyset$
$< FF >$	::=	ff, | $\emptyset$
$< GG >$	::=	gg, | $\emptyset$

feature from the original feature-set. Then we use our chunking algorithm on this new set of feature-sets to generate candidate merges. This gives us both an equivalence class ontology and the contexts for which these are equivalent.

To illustrate this, we artificially generated 9,872 feature-sets by the sampling from the grammar in Table 3.4.

We sampled from this 10,000 times, and eliminated duplicates, which gave us 9,872 items, a sampling of which are shown below.

{aa, bb, cc, dd, ee, 4B, 5D, 6D, 7B}	{aa, bb, ee, gg, 4C, 5A, 6D, 7A}	{bb, ee, gg, 4C, 5E, 6F, 7B}
{aa, cc, ee, ff, 4A, 4B, 5B, 6B, 7B}	{cc, dd, gg, 4C, 5A, 6B, 7B}	{aa, bb, dd, 1F, 2A, 3A}
{aa, bb, dd, ff, gg, 1C, 2D, 3F}	{bb, cc, ff, gg, 1E, 2C, 3F}	{cc, ff, 1B, 2A, 3B}
{bb, cc, dd, ee, 4C, 5D, 6B, 7B}	{cc, dd, ee, 4C, 5A, 6C, 7B}	{ff, 4B, 5E, 6E, 7A}
{bb, cc, dd, gg, 1F, 1E, 2B, 3D}	{aa, dd, ff, 4A, 5E, 6C, 7B}	{1A, 2A, 3F}

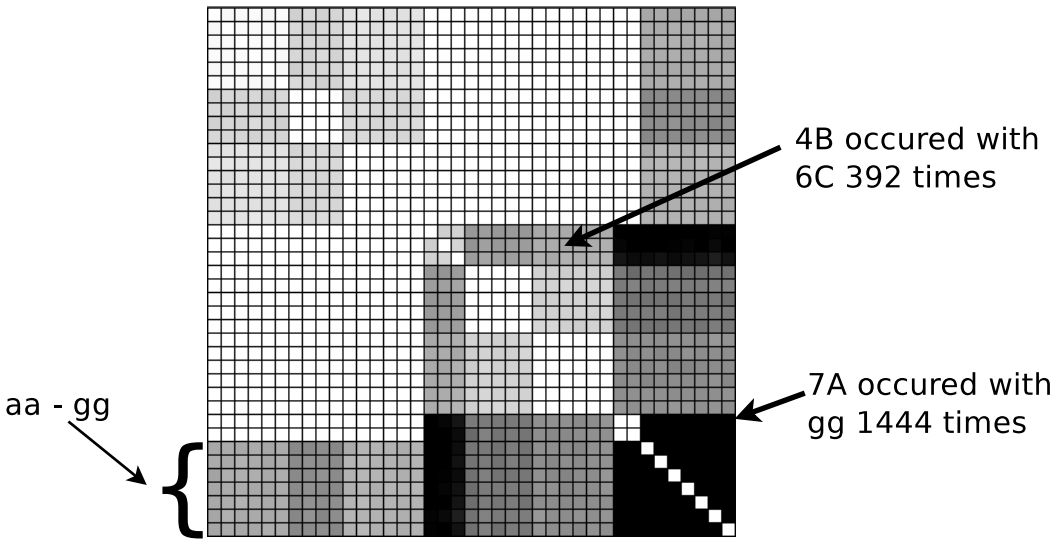


FIG. 3.14. **The matrix formed from 9,872 samples.** Entry i, j is the number of times feature i co-occurs with feature j. White is 0 times and black is 1,000 times or more, with shades of grey for values between.

Now our goal is to get back the original 8 equivalence classes, given only these unordered sets.

To do this, we create "context" vectors for each feature. That is, we create a co-occurrence matrix where entry i, j is the number of times feature i co-occurs with feature j. The 39 by 39 matrix for our features is shown in Figure 3.14.

We then treat the rows of this matrix as feature-sets and chunk these. This gives us the ontology in Figure 3.15. Not only does this give us candidate equivalence classes, but it also provides us with the common context under which these features are interchangeable. Note that the features `4A`, `4B`, and `4C` form an equivalence class under node `N00003`. Not shown in this figure is the context inside node `N00003`, which says that each of these nodes occurs with `5A` at least 432 times, `5B` at least 393 times,

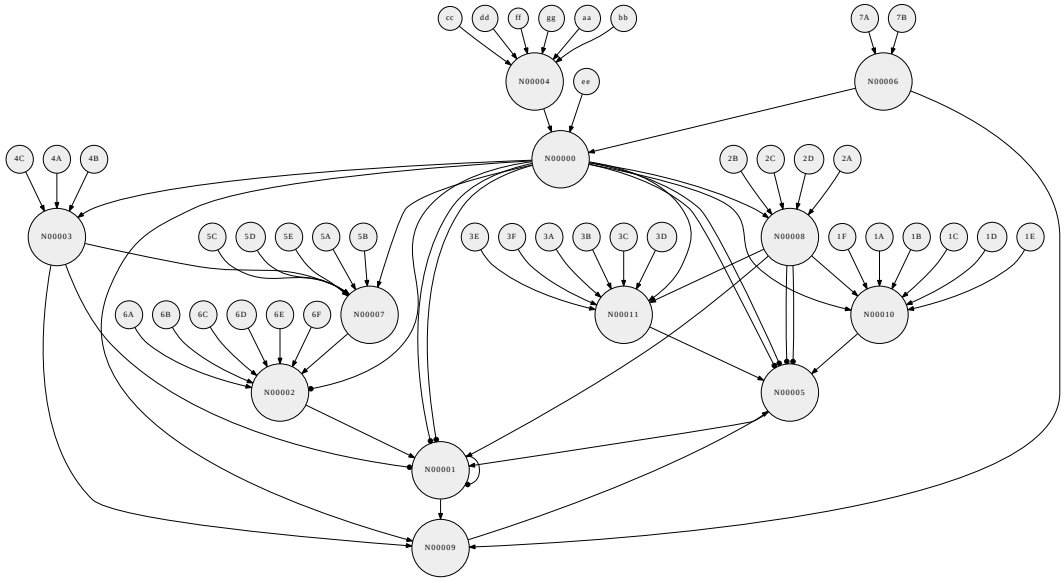


FIG. 3.15. **The results of chunking context.** We extract equivalence classes by treating the ancestors of each node as an equivalence class. These correspond to the equivalence classes in the grammar in Table 3.4.

`7B` 1061 times, etc..

As another example, suppose we have 16 equivalence classes with 4 members each, and we pick pairs of these equivalence classes, but not uniformly. That is, the pair C_0, C_3 might be likely, but C_0, C_2 not. If we have 400 unique samples, the co-occurrence matrix will be as shown in Figure 3.16. As shown, we can see not only the equivalence classes, but also which equivalence classes are co-occur with each other. Given this matrix, we can use methods similar to (Kemp & Tenenbaum 2008) to discover the classes.

Ideally, the elements in an equivalence class should be mutually exclusive. For example, if `A` occurs if and only if `B` occurs, then these two features will have exactly

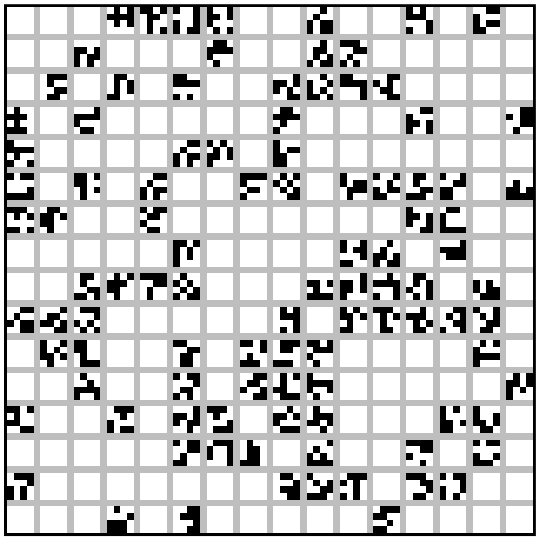


FIG. 3.16. **The matrix formed from 400 samples.** There are 16 equivalence classes with 4 members each.

the same vector. However, if we ran chunking on the original set of feature-sets, this should have already created a new chunk (called `C` say) of `A` and `B`. Then we'll only have a single instance of `A` or `B`, namely in `C`, and `C` will have the "context" feature-set. So, if `A` and `B` co-occur together a lot, chunking should have already chunked them, so every place where they co-occur gets replaced by the symbol for the chunk.

To make this explicit, we show our merging algorithm in Table 3.5. We've shown as proof-of-concept that chunking contexts can be used to discover equivalence classes. We leave for future work combining Chunking and Merging to create a single ontology with ANDs and ORs. Since merging is simply chunking contexts, if we have a good chunking algorithm, this will spill over to our merging algorithm.

Also note that by chunking context, if we have 2 nodes in an equivalence class, the context under which these nodes are equivalent is found simply bind finding the

Table 3.5. **The Merging algorithm**

```
// Returns a set of candidate equivalence classes.
// S is an ontology. This ontology could be "flat" (i.e., raw feature-sets)
// or a heterarchical conceptual structure.
Merger (S)
  // Find the cooccurrence tallies t(a, b) for every pair of nodes in S
  foreach top-level "instance" node i ∈ S
    create i's "flattened" set flat(i) by doing the following
      turn i ON
      use the truth-derivation algorithm to find what other nodes turn ON
    foreach pair of elements (a, b) ∈ flat(i) × flat(i)
      increase the tally t(a, b) by 1. (This tally is initialized to 0.)
  // Now create the "context" vectors from the tallies
  let F = ∅ // F is an empty ontology
  foreach node n ∈ S
    create a feature-set f_n, where the feature c_i for f_n is t(n, i)
    let F = F ∪ f_n // Add f_n to F
  // Crunch the "context" vectors
  let M = parentCrunch (F)
  // Now, extract the equivalence classes
  let EQClasses = ∅
  foreach node m ∈ M
    // a is an ancestor of m iff m is a descendent of a
    let A(m) be the ancestors of m
    let EQClasses = EQClasses ∪ A(m)
  return EQClasses
```

nodes' common ancestor and "flattening" that node (i.e., finding its descendent sensor nodes). For example, in Figure 3.15, nodes 4A, 4B, and 4C all point to node N00003. For clarity, the descendents of node N00003 aren't shown in the figure, but these include nodes 6C, because 4A, 4B, and 4C all cooccur with node 6C with around the same frequency.

Ideal Energy Cost for Chunking and Merging The following theorem is given to help guide when Merging saves us description length.

Theorem 5. *If we have 2 equivalence classes with m items each, where there are p*

instances of each of the m^2 items (and each item is equally likely), and we assume an ideal Huffman encoding to specify each item (where each item takes n bits initially), it will be more cost effective to merge than to crunch the items if $n < 2\log_2(m)$.

Proof. The number of bits needed to specify the pm^2 pairs for naive, crunched, and merged encodings are:

$$\begin{aligned} NaiveCost(m,n,p) &= 2pnm^2 \\ CrunchedCost(m,n,p) &= pm^2(n+\log_2(m)) + nm^2 \\ MergedCost(m,n,p) &= pm^2(n+\log_2(m)) + 2nm + 2n + 2\log_2(m) \end{aligned}$$

So, we choose merge over chunk when:

$$\begin{aligned} pm^2(n+\log_2(m)) + nm^2 &> pm^2(n+\log_2(m)) + 2nm + 2n + 2\log_2(m) \\ nm^2 &> 2nm + 2n + 2\log_2(m) \\ m^2 &> 2m + 2 + \frac{2\log_2(m)}{n} \\ m^2 - 2m + 1 &> 3 + \frac{2\log_2(m)}{n} \\ (m-1)^2 &> 3 + \frac{2\log_2(m)}{n} \\ m &> 1 + \sqrt{3 + \frac{2\log_2(m)}{n}} \end{aligned}$$

If we let $e = \sqrt{3 + \frac{2\log_2(m)}{n}} - \sqrt{3}$ (a positive, but likely small value), then

$$
\begin{aligned}
m &> 1 + \sqrt{3 + \frac{2\log_2(m)}{n}} - \sqrt{3} + \sqrt{3} \\
m &> 1 + \sqrt{3} + e \\
m &\gtrapprox 2.732 + e \\
m &\gtrapprox 3
\end{aligned}
$$

So, as long as we have 2 equivalence classes, roughly equally distributed with at least 3 items in each, it should be more cost effective to merge than to crunch the items.

For our approximation to be broken, the following must hold:

$$
\begin{aligned}
\sqrt{3 + \frac{2\log_2(m)}{n}} &> 2 \\
2\log_2(m) &> n
\end{aligned}
$$

If we have 1,024 features, then n will be 10, so m would have to be $2 \cdot 2^5$ or 64, which is unlikely for such a small number of features. □

Note that under this framework, forming a single equivalence class by itself yields no savings.

3.6 Application to Supervised Learning

Any intelligent system will need to be a lifelong learner. That is, when encountering new tasks, any intelligent system should be able to draw upon background knowledge constructed from years' worth of experience.

Standard supervised learning systems don't have this background knowledge upon which to draw. Compared to standard supervised learning algorithms, people require remarkably few training instances to learn a concept. For example, consider the "adze" shown in Figure 3.17. We assume that shown only the leftmost image (of the man holding the adze) and being told "This is an adze.", most English-speaking adults would be able to correctly identify an adze from a group of images. Given all 3 images, it's doubtful that any person would have problems correctly identifying this tool.

There are a few points to note. First, we assume that people can learn the adze concept from these few images *even if they've never seen an adze before.* That is, it isn't the case that we have the adze concept, then just need to name it. Yet, we somehow know that the adze is the tool the man's holding. Second, we can learn this concept in the absence of negative examples. That is, we are shown no images of which we are told "This is *not* an adze.". Somehow, people use their existing conceptual structures to learn from so few examples. We propose to shed light on some of these questions by showing how we can use conceptual structures developed by Ontol to likewise learn from a handful of positive examples.

A final note (that our model doesn't yet directly address, but which we also hope to shed some light on), is that we assume that we can learn the adze concept given

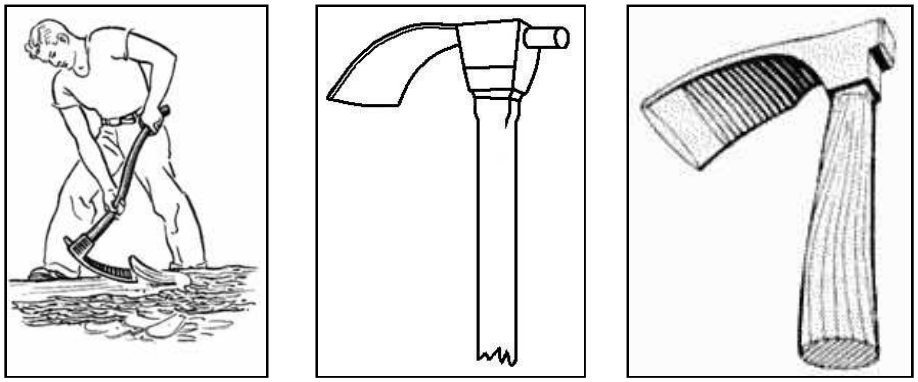


FIG. 3.17. **An adze.**

only the image on the left (with the man). In this case, the adze isn't the only thing in the image. Conceivable, the concept of "adze" could mean the man's particular stance, the type of split in the log, the man's name, or the type of shirt the man's wearing.

We demonstrate the utility of the ontology learned in Ontol by introducing a new algorithm for semi-supervised learning, which uses ontologies learned with Ontol on tasks where we're given a large set of unlabeled instances and only a handful of labeled instances, where these instances might all be positive examples. It should be noted that our focus was never on semi-supervised learning, but this area was a simple extension of our system.

3.6.1 Related Work

The field of Semi-supervised Learning addresses how unlabeled data may be used to improve performance in the scarcity of labeled data. For a survey of Semi-

supervised Learning, see (Zhu 2008). Generally, Semi-supervised learning systems extract features from the unlabeled data, then feed these features (for the labeled data) into a standard learning algorithm such as Support Vector Machines (SVMs). This approach, however, requires negative examples to train the SVM. This approach also typically assumes that the labeled data comes from the same distribution as the labeled data. For example, this approach would assume that we've seen several adzes before being told what an adze is.

To address this latter problem, (Raina *et al.* 2007) introduces the idea of Self-taught Learning. This algorithm attempts to extract features that are useful in general. For example, the algorithm extracts useful features from images, such as edge detectors, then puts these features into a SVM to learn new images. However, this approach also requires negative examples. From this work, we suspect that the concepts developed by Ontol, such (such as contiguous "chunks" of pixels for image data) can be used to increase learning efficiency.

Both Semi-supervised learning and Self-taught learning not only require both positive and negative labeled instances, but both typically require tens if not hundreds of labeled instances. We know of little work on Semi-supervised learning from only a handful of training instances. One exception is (hua Zhou, chuan Zhan, & Yang 2007), which claims to learn from a single labeled positive example. This algorithm basically uses a distance metric to find similar unlabeled instances. However, this distance metric is not learned from the unlabeled instances, so it must be modified by the user for each domain.

There has been work on learning from only positive instances. For example, both (Elkan & Noto 2008) and (Meraz *et al.* 2004) assign a probability that unlabeled

examples are positive or negative given the distance from labeled nodes, but these distance metrics aren't learned from unlabeled examples, so they must be modified by the user for each domain. These approaches also typically need hundreds of positive instances to make accurate classifications. The idea of "one-shot" learning has been explored in the work of (Fei-Fei, Fergus, & Perona 2006), which shows how visual domain knowledge can be used with Bayesian methods to learn object categories from a few positive instances. However, the implementation in this approach is limited to visual domains.

There are Inductive Logic Programming algorithms that can learn from only positive data, for example (Muggleton 1996). This algorithm works by finding the "least general generalization". That is, it finds the theory that includes all the positive examples, but as few of the remaining instances as possible. However, this doesn't begin by making a general representation from the unlabeled data that's useful across several categorization tasks. Instead, it uses only bottom-level features.

3.6.2 Semi-supervised Learning using an Ontology

Here we describe our algorithm for using an ontology created by Ontol to learn from a handful of positively labeled examples. The basic idea behind the algorithm is searching for theories to maximize a Bayesian energy function.

An Energy Function for Semi-supervised Learning Given an ontology Ω, a set of unlabeled data U, and a set of labeled data L, we seek to construct a Boolean expression M consisting of the terms M_i, where each term M_i is an element

of our ontology Ω. We assume that M is true for every positive instance (i.e., when we run our truth-derivation algorithm from Section 3.4 on the instance, M is true) and M is false for any negative instances we're given. There are many such expressions, so we search for the M that minimizes for the value in 3.4 below.

$$E(M) = |D| \log_2 N(M) + |M| \log_2 |U| - \sum_{M_i \in M} \log_2 N(M_i) \tag{3.4}$$

where $N(M_i)$ is the number of items in the *unlabeled* data U for which M_i is true, and $N(M)$ is the number of items in the unlabeled data U for which $M = true$.

Intuitively, $\left(|M| \log_2 |U| - \sum_{M_i \in M} \log_2 N(M_i)\right)$ is the description length of the model, and $(|D| \log_2 N(M))$ is the description length of the data using the model.

Theorem 6. *If we assume a description-length prior on our model, then Equation 3.4 maximizes the probability of our model M given an* observation *D.*

Proof. If we assume that we construct M such that for every $D_i \in D$, M is true if D_i is true (i.e., $P(M = true|D_i) = 1$), then

$$P(D_i|M = true) = \frac{P(M = true|D_i)\, P(D_i)}{P(M = true)} = \frac{\frac{1}{|U|}}{\frac{N(M)}{|U|}} = \frac{1}{N(M)}$$

Using this, our equation to maximize the probability of M given D becomes

$$\begin{aligned}
\arg\max_M P(M|D) &= \arg\max_M \frac{P(D|M)\,P(M)}{P(D)} \\
&= \arg\max_M \left(P(D|M)\,P(M)\right) \\
&= \arg\max_M \left(P(D|M) \prod_{M_i \in M} \frac{N(M_i)}{|U|}\right) \\
&= \arg\max_M \left(\prod_{D_i} P(D_i|M = true) \prod_{M_i \in M} \frac{N(M_i)}{|U|}\right) \\
&= \arg\max_M \left(\log_2 \prod_{D_i} P(D_i|M = true) \prod_{M_i \in M} \frac{N(M_i)}{|U|}\right) \\
&= \arg\max_M \left(\sum_{D_i} \log_2 P(D_i|M = true) + \sum_{M_i \in M} \log_2 \frac{N(M_i)}{|U|}\right) \\
&= \arg\max_M \left(-|M| \log_2 |U| + \sum_{D_i} \log_2 P(D_i|M = true) + \sum_{M_i \in M} \log_2 N(M_i)\right) \\
&= \arg\min_M \left(|M| \log_2 |U| - \sum_{D_i} \log_2 P(D_i|M = true) - \sum_{M_i \in M} \log_2 N(M_i)\right) \\
&= \arg\min_M \left(|M| \log_2 |U| - \sum_{D_i} \log_2 \frac{1}{N(M)} - \sum_{M_i \in M} \log_2 N(M_i)\right) \\
&= \arg\min_M \left(|M| \log_2 |U| + |D| \log_2 N(M) - \sum_{M_i \in M} \log_2 N(M_i)\right)
\end{aligned}$$

which is the same as minimizing Equation 3.4. □

Searching to Minimize Equation 3.4 We seek to find a Boolean expression M to minimize Equation 3.4. We do a lattice-like search similar to Inductive Logic Programming (ILP) (Muggleton 1994). Our search is bidirectional in the sense that it both refines the search and relaxes the search. Refinement is done by adding

to existing conjunctions, and relaxation by adding new conjunctions to our existing theory using an OR. For simplicity, our Boolean expression is a disjunction of conjunctions. Essentially, we initialize our expression M to the disjunction that contains our positive instances, then we hillclimb on Equation 3.4. We generate variants of M by generalizing by including the parents of each node as new disjunctions in our expression, and by adding nodes' parents to existing conjunctions. Our semi-supervised learning algorithm doesn't need negative instances, but it can make use of them if they're provided. So, if there are negative instances and a new version of M contains negative instances as members of the class for which M is *true* or doesn't contain positive instances, then we throw that version out. A more detailed algorithm is given in Table 3.5.

3.6.3 Experiments

We tested our Semi-supervised learning algorithm on several datasets from the UCI Machine Learning Repository (Blake & Merz 1998). For each dataset, we tested classification using a range of positive and negative examples. If a datasets had $n > 2$ classes, we treated the dataset as n separate binary class problems. We created a baseline by giving the accuracy if every item is classified as the most frequent class for the dataset. For comparison, we show results for our algorithm with the exception that The Cruncher isn't run. This modified algorithm is the same as that given by (Muggleton 1996). No other Semi-supervised learning algorithms work with so few positive examples. The results are shown in Table 3.7, in which categories are learned using only 5 positive instances and no negative instances, and Table 3.8, in which we show the effects of increasing the number of positive instances and adding negative

Table 3.6. **The Semi-supervised Learning algorithm.**

// Given unlabeled data U, positive instances P, negative instances N
// Return set of $u \in U$ such that u is predicted to be positive.
// (For shorthand, we define $M(u)$ to be true if when we parse
// u using ontology Ω, M is ON.)
semiSupervisedLearning (U, P, N)
  // Crunch to get the ontology Ω, using algorithm from Table 3.1.
  let Ω = Cruncher(U)
  // Now search for M to maximize Equation 3.4.
  parse the positive instances P according to Ω, and add them to the ontology
  // Initialize M to be the disjunct of the positive instances.
  let $M = \bigvee_{p \in P} p$
  let $openNodes = \{M\}$
  let $closedNodes = \{\}$
  // Hill climb
  while $|openNodes| > 0$ **and** some number of iterations hasn't been reached
    // now try variants of M by adding parents.
    Randomly choose L from $openNodes$ and remove it, and add it to $closedNodes$.
    // "open" L by generating all variants,
    $variants$ = open(L, P, N)
    // add these to openNodes, if they're not already in $openNodes$ or $closedNodes$.
    let $openNodes = openNodes \cup (variants - closedNodes)$
    evaluate each variant using Equation 3.4, and keep track of the model that minimizes this Equation.
  // Return all u that turn ON M
  return $\{u \in U | M(u) \, istrue\}$

// To "open" a Boolean expression L, add its parents to the
// disjuncts and to each conjunct. P and N are the positive
// and negative instances.
open(L, P, N)
  $variants = \{\}$
  foreach *parent* in all parents of nodes in L
    create L_d by adding that parent to L as a new disjunction
    for all old disjunctions j in L_d
      // A disjunction "covers" another if the set that turns ON the
      // former is a superset of that for the latter.
      if the new disjunction completely covers another disjunction
        then remove the covered disjunction from L_d
    foreach existing conjunction c of L
      create L_{pc} by adding that parent to c
  add all L_ds and L_{pc}s to $variants$
  remove any $l \in L$ for which $\exists p \in P | l(p) = false$
  remove any $l \in L$ for which $\exists n \in N | l(p) = true$
  return $variants$

instances. The data was generated by running the following process 100 times: For each data set, a class was chosen at random, and p positive instances and n negative instances were found from that class. These, along with the unlabeled data, were then fed to the respective classifiers.

From this table we can see that Ontol is competitive with the ILP algorithm, being statistically significantly better for 2 of the datasets (connect-4 and zoo), while never being statistically significantly worse than ILP.

Both ILP and Ontol performed worse than the Baseline for the connect-4, SPECT, and tic-tac-toe datasets. We suspect that this is because such a tiny sample size (5) with no negative examples could easily lead to overgeneralization of positives, yielding a high percentage of false positives (Type I errors), which is what we see.

Ideally, the labeled examples should have a significant amount of breadth. For example, if we want to learn the concept of "mammal", our exemplars shouldn't be 3 different dogs. In that case, our learner would be more likely to come to the conclusion that all mammals are dogs. Our exemplars were chosen randomly. So this situation —a too narrow selection of exemplars— happened occasionally. We speculate that when people give other people examples, there might be an implicit assumption that the teacher will give training examples that won't lead to an overspecialized theory.

Conversely, we would also expect our training examples not to share some rare property that we're not interested in. For example, if our training instances for "mammal" are a cow, a dog, and a hamster (all of which are domestic, a relatively rare property among animals), we might wrongly infer that a chicken is also a mammal because it's domestic.

Table 3.7. **Results for Semi-supervised Learning.** Various UCI datasets are shown with training on 5 positive instances (and no negative instances). The average percentage accuracy is shown for "baseline" (classifying every item as the most frequent classification), Muggleton's ILP algorithm and our "Ontol" algorithm. The 95% confidence interval is shown for the means. Also shown are the average percentage of Type I and Type II errors. Note that these are unnecessary for the Baseline case because every error for Baseline is a false positive.

		ILP			Ontol		
Dataset	**Baseline**	**Avg.**	**TI%**	**TII%**	**Avg.**	**TI%**	**TII%**
connect-4	76.13%	46.67±2.34%	39.18%	14.14%	50.69±2.45%	33.21%	16.10%
house-votes-84	61.38%	82.48±2.70%	7.03%	10.50%	83.14±2.08%	5.79%	11.06%
kr-vs-kp	53.10%	56.49±1.71%	27.37%	16.14%	54.65±1.32%	25.60%	19.75%
mushroom	51.40%	71.00±2.27%	12.06%	16.94%	72.38±2.05%	8.36%	19.27%
nursery	67.85%	72.03±3.40%	21.23%	6.73%	71.72±3.63%	20.86%	7.42%
SPECT	79.40%	56.84±1.30%	27.16%	16.00%	57.37±1.50%	24.99%	17.64%
tic-tac-toe	65.34%	53.85±1.41%	23.12%	23.03%	53.58±1.80%	21.09%	25.33%
zoo	83.03%	83.09±4.15%	16.71%	0.20%	92.59±1.74%	6.95%	0.46%

In Table 3.8, we show the effect of increasing the training size (and adding negative examples) for learning the Zoo dataset. As we'd expect, increasing negative examples decreases the number of false positives. Also note that Ontol consistently beat out ILP, and both were able to get nearly perfect accuracy from only 40 labeled instances.

As to why Ontol works so well, we suggest that a node is created in our ontology *because* it's useful for characterizing the data. We assume that concepts that people develop or might be interested in are likewise useful for characterizing our world.

The nodes in our ontology bias the search. For example, the description length of hair, milk, NO-eggs is length 1 given our ontology. Without the ontology, it's 3. (Note that the cost of specifying each node is a little more expensive if there are more nodes though, so the ontology *can* hurt us.)

Table 3.8. **Results for Semi-supervised Learning on the Zoo dataset for different training set sizes.** The percentage of false positives (Type I error) is also shown.

		ILP			Ontol		
#Pos	**#Neg**	**Avg.**	**TI%**	**TII%**	**Avg.**	**TI%**	**TII%**
5	0	83.09±4.15%	16.88%	0.20%	92.59±1.74%	7.02%	0.46%
10	0	86.67±3.81%	13.46%	0.00%	91.70±1.85%	8.38%	0.00%
20	0	86.87±3.61%	13.26%	0.00%	92.70±1.64%	7.37%	0.00%
5	5	93.08±2.17%	6.66%	0.33%	95.17±1.75%	4.39%	0.49%
10	5	92.79±2.44%	7.28%	0.00%	94.93±1.79%	4.97%	0.15%
20	5	94.25±1.89%	5.81%	0.00%	95.38±1.74%	4.67%	0.00%
5	10	96.01±1.10%	3.10%	0.93%	96.91±0.88%	2.32%	0.80%
10	10	97.43±0.78%	2.54%	0.06%	98.34±0.63%	1.48%	0.20%
20	10	97.15±0.97%	2.88%	0.00%	97.52±0.88%	2.50%	0.00%
5	20	97.68±0.57%	1.08%	1.26%	98.64±0.45%	0.74%	0.63%
10	20	98.56±0.44%	1.18%	0.27%	99.06±0.36%	0.88%	0.07%
20	20	98.93±0.50%	1.08%	0.00%	98.78±0.51%	1.23%	0.00%

Also, it has not escaped our attention that there is considerable debate about the role negative examples play in child language acquisition (Marcus 1993), (Rohde & Plaut 1999). Gold (Gold 1967) and Chomsky (Chomsky 2005) argue on theoretical grounds that there must be some innate Universal Grammar or bias in the human brain to allow humans to learn language from so few examples and in the absence of negative examples. This argument is referred to as "Poverty of The Stimulus" (POTS). Our semi-supervised learning model may lend some insight into how vocabulary might be learned from only positive examples. We leave as future work how these ideas may also extend to grammar acquisition. We speculate that Chomsky's bias may be nothing more than some variant of an information theoretic derivable evaluation function, fundamentally the same as Ockham's Razor.

3.6.4 Discussion

In this Chapter, we've introduced Ontol, which builds a useful conceptual structure from raw feature-set data. We've demonstrated the utility of the concepts learned by Ontol through a compression algorithm, a Reinforcement Learning algorithm, and a Semi-supervised learning algorithm. But we believe the value of Ontol lies beyond these "spin-off" algorithms. We mean for Ontol to be the backbone of a full cognitive system. Since it's knowledge-free, it's applicable to a wide range of domains. Furthermore, if a relational domain can be expressed as a collection of feature-sets, then the algorithms for ontology formation, inference, and constraint satisfaction search can be applied to these systems. For example, if actions and goals can be phrased as constraints, then planning can be implemented as a constraint satisfaction problem.

Chapter 4

PARAMETERIZED CONCEPTS

> Propaganda, slavery, decoding, entrapping, mimicry, panhandling, Trojan horses, highwaymen, cuckoos: they are all present among the ants and the predators and social parasites that victimize them. Such words may seem unduly anthropomorphic, turning ants and their associates into little people. But perhaps not. It is equally possible that the number of social arrangements available to evolution anywhere in the world, or even in the universe, is such that the phenomena we have recounted here are inevitable natural categories of exploitation wherever it may occur.
>
> –From "Journey to the Ants" (Hölldobler & Wilson 1998)

This chapter goes into detail about how we can find behaviorally similar cortical regions and use these to compress our ontology, generalize knowledge, and discover invariant concepts. These algorithms assume that we already have an AND/OR ontology created by Ontol.

Recall from Subsection 2.2.2 that we can't get translation invariance from chunking and merging alone. We need isomorphisms among cortical regions to get this. Furthermore, because different structures can have similar behavior (in terms of how truth-settings for nodes influence each other, as discussed in Subsection 2.2.2), we need *behavioral* isomorphisms. There are several open questions for this. The first question we attempt to answer is how parameterized concepts can be represented in the same framework as Ontol. We provide a proof of concept for how this might be done. Secondly, there's the question of how behavioral isomorphisms can be found efficiently. That is, how can we recognize whether 2 cortical regions are behaviorally similar to each other. We show a proof of concept of how this might be done by introducing "behavioral signatures" for cortical areas. A third question that we leave for future work is how the cortical regions can be segmented in the first place. That is, the ontologies produced by ontol are usually connected in the graphical sense (i.e., there's a path from each node to every other), so the cortical regions are rarely neatly segmented. This part, finding and representing parameterized concepts, is an essential span of the bridge from raw-sensors to a rich relational theory of the world. It feeds into Phase 3, and ultimately to Phase 4, which completes our story of how a system might go from raw sensor data to a theory rich enough to have an "intuitive" theory of numbers.

As described in Subsection 2.2.2, it's possible for 2 cortical regions to have similar behavior, but different structure. Such regions are "behaviorally similar", but not "structurally similar". Since our cortical regions are autonomously constructed, only the regions' behavior matters. Therefore, we're interested in finding behaviorally similar cortical regions. We define a mechanism and representation by which behaviorally similar cortical regions can be parameterized and called, and we provide a

heuristic "signature" by which the behavioral similarity of different cortical regions can be efficiently measured. Using these, we can create and use parameterized cortical regions to further reduce the description length of our ontology.

4.1 Related Work

Though we know of no work that extracts *behavioral* similarity from different structures, there has been work on analogy, graph isomorphism, and graph grammars. These are reviewed below.

4.1.1 Analogy

O'Donoghue breaks analogy into 5 phases (O'Donoghue 2005): *representation*, *retrieval*, *mapping*, *validation*, and *induction*. *Representation* deals with how knowledge and analogies are represented. *Retrieval* concerns finding where an analogy might occur. *Mapping* is the process of finding which parts of a pair of situations or domains correspond to each other. *Validation* is the process by which one determines how valid an analogy is. Finally, *induction* is the process by which general concepts are learned by the analogy. For example, if one notices that the situation where an ant forcibly takes a beetle corpse from another ant is analogous to the situation where a person forcibly takes another person's wallet, then one can create a general concept from this analogy (possibly called "robbing").

In this 5 phase terminology, most previous work on analogy has focused on the *mapping* phase (Marshall & Hofstadter 1996), (Falkenhainer, Forbus, & Gentner

1989). Though there has also been work on *retrieval* (Forbus 2001), (O'Donoghue 2005), most of this work has assumed that the data has already been pre-segmented into domains of relatively small size. This requires either a segmentation algorithm, or (more likely) a human expert. Furthermore, all of these algorithms are for structural similarity, not behavioral similarity. An overview of many of these systems can be found in (O'Donoghue 2005). It has been argued from theoretical grounds that analogy should bring about conceptual change (Dietrich 2000), but we know of few existing algorithms that do analogical induction. One exception is the LISA algorithm (Hummel & Holyoak 2006) which, however, must be provided with the domains for analogizing, and also works only on domain structure, not behavior. We will necessarily also address issues of *representation* and *mapping*. As for *validation*, the driver in our system is description length, so a concept is useful if it best reduces description length.

4.1.2 Graph Isomorphism

Finding analogies is structurally similar to finding isomorphisms between and within graphs. It's conjectured that there's no polynomial time algorithm for determining whether a pair of arbitrary graphs are isomorphic (Skiena 1990), although some special cases have faster algorithms (Skiena 1990), (Hopcroft & Wong 1974). In practice, the Nauty algorithm (McKay 1981) efficiently finds isomorphisms within a graph. It does this using a number of time-saving heuristics, many of which can be useful for our purposes.

The SUBDUE system (Holder, Cook, & Djoko 1994) compresses graphs by finding common substructures. Unlike many other analogy systems, the data in SUBDUE

is not assumed to be pre-segmented, and minimum description length is the guiding principle by which substructures are evaluated. Furthermore, SUBDUE does *induction* in the sense that frequently occurring substructures are replaced by a node that symbolizes the full substructure. Although SUBDUE and other systems don't do *behavioral* isomorphism, if we encode behavior as a graph, we can throw our data at some of these systems.

4.1.3 Graph Grammars

Since the data is unsegmented, classification is akin to parsing a graph given a graph grammar, and building the ontology is akin to learning a graph grammar. Work has been done on estimating parameters of a graph grammar (Oates, Doshi, & Huang 2003), but there has been little work on learning the structure of graph grammars.

4.2 Mechanics of Parameterized Calls

This section describes how parameterized cortical areas are represented and used.

Suppose we have two cortical regions that are behaviorally isomorphic, such as those shown in Figure 4.1. If we're only interested in the behavior of these graphs in terms how their bottom inputs affect each other's ON/OFF value, we can extract out the AND/OR structure above these and *parameterize* the differences. For this example, we note that, in the left network, input nodes C, D, X, Y, Z, and W play the role of to E, F, K, L, M, and N, in the right network respectively (actually nodes

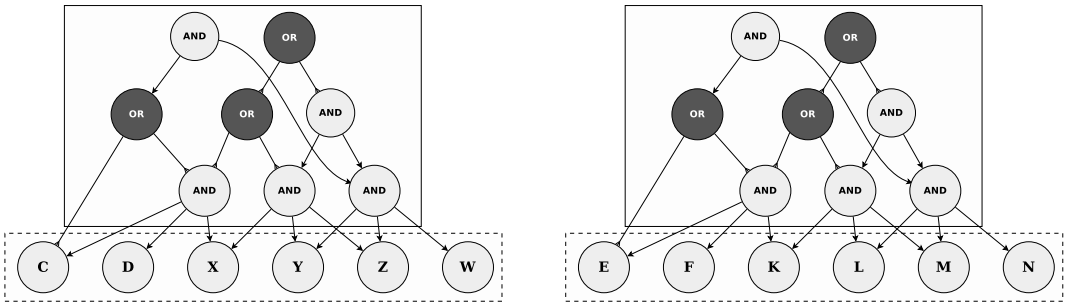


FIG. 4.1. **Two isomorphic cortical regions.** These cortical regions happen to be both behaviorally and structurally isomorphic.

Y and Z are interchangeable with each other, as are nodes L and M).

Figure 4.2 shows how we represent this parameterization. To do this, we introduce a new kind of node, called a BIND node that, when ON, causes the values of all its children to be the same value.

If a BIND node is OFF, it has no effect on the values of its children. If a BIND node with 2 children is ON, and one of its children is UNSPECIFIED, the BIND node causes this child to be the same value as the other child. It should be rare that a BIND node is ON with one child ON and the other OFF. If this happens, the BIND node won't change either value, but should raise an error because something has gone wrong. More formally, if A is a BIND node that has children C_1 and C_2, then the values of C_1 and C_2 are modified according to the following cases:

1. If A is OFF, then the values of C_1 and C_2 are unaffected.

2. If C_1 and C_2 have the same values, then no nodes change values.

3. If A is ON, C_1 is ON or OFF, and C_2 is UNSPECIFIED, then C_2 is set to ON

or OFF, respectively.

4. If A is ON, C_2 is ON or OFF, and C_1 is UNSPECIFIED, then C_1 is set to ON or OFF, respectively.

5. If A is ON, C_1 is ON or OFF, C_2 is ON or OFF but not the same value as C_1, then an error is raised.

For example, in Figure 4.2, the OR node "B" acts as a gate for which set (I ($= \{C, D, X, Y, Z, W\}$) or J ($= \{E, F, K, L, M, N\}$)) is bound to set R ($= \{1, 2, 3, 4, 5, 6\}$). If B is ON, only *one* of I or J need be ON, so for this example, I and J are mutually exclusive.

If B is OFF, then I and J will both be off. This will leave nodes 1, 2, 3, 4, 5 and 6 all UNSPECIFIED.

Suppose that C, D and X are all OFF, Y and Z are both ON, and that W is UNSPECIFIED. Suppose we then turn B ON and instantiate B by turning I ON. The BINDs will cause nodes 1, 2, and 3 to all go from UNSPECIFIED to OFF. Likewise, nodes 4 and 5 will turn ON, and 6 will remain UNSPECIFIED. At this point, the rightmost AND will turn ON to explain nodes 4 and 5. (The middle AND won't turn ON because it'd then have to explain why 3 is set to OFF.) Because 6 is a child of the rightmost AND, which is now ON, node 6 will also turn on. Because there's an ON BIND node that connects to both 6 and W (and W is UNSPECIFIED), W will then be set to ON.

Note that I and J form an *equivalence class* because either can be bound to R.

We can also have "split" cortical calls, where not all the inputs are tied together.

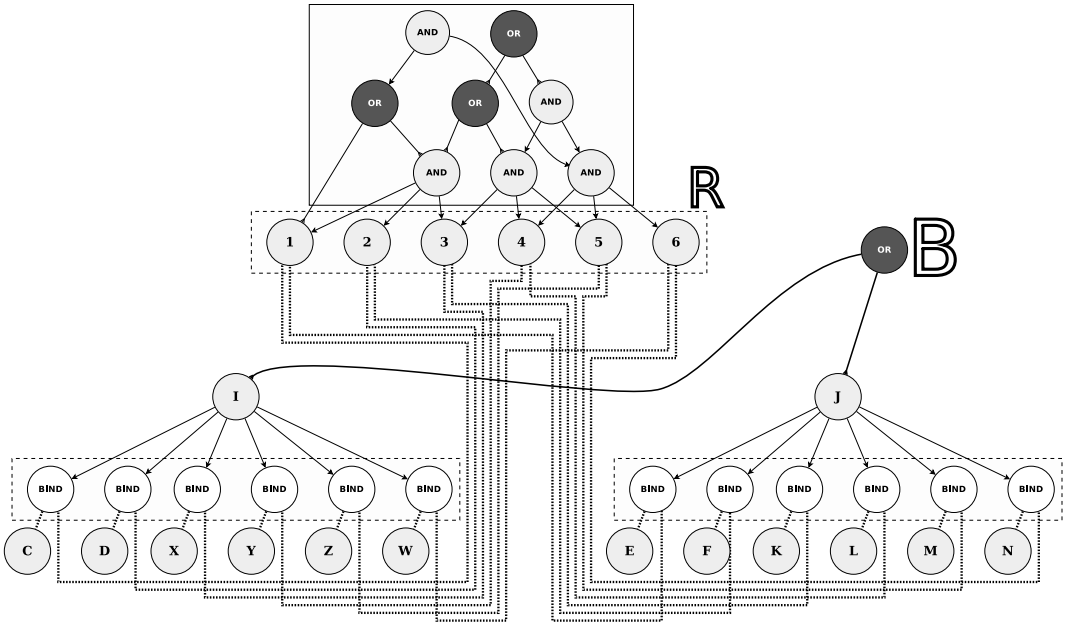


FIG. 4.2. **An example call graph, unifying the 2 graphs from Figure 4.1.** The OR node "B" gates which set is bound to R to "call" the common cortical area.

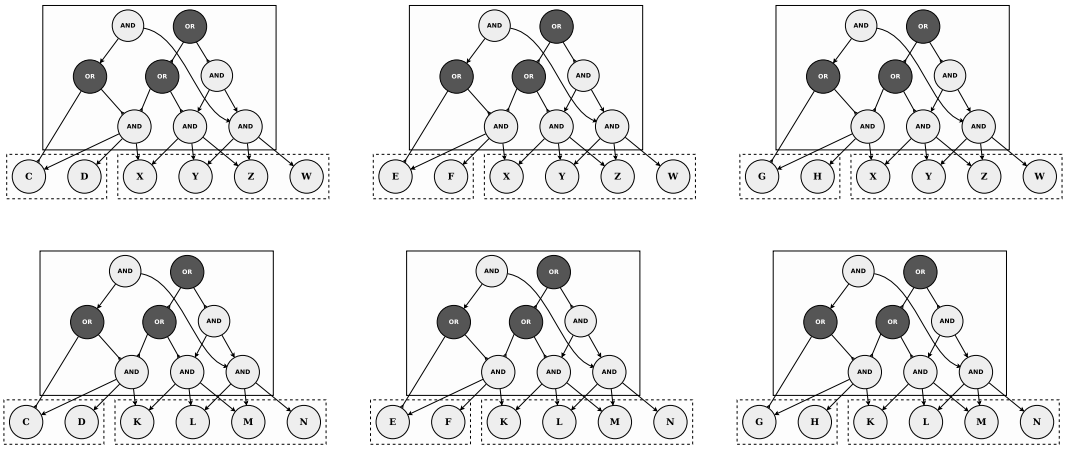


FIG. 4.3. **Six isomorphic cortical regions.**

For example, instead of only 2 isomorphic cortical regions, Figure 4.3 shows 6 isomorphic regions. Note that, in the top row, the 4 rightmost nodes are all X, Y, Z, and W, whereas in the bottom row, these are K, L, M, and N. Similarly, in the 1st column the leftmost 2 nodes are C and D. The same for E and F in the middle column and G and H in the rightmost column.

Figure 4.4 shows how we can parameterize the cortical regions in Figure 4.3. Here the input to the common region is "split". In this figure, S, T, and U form an equivalence class for possible parameters for Q (the nodes 1 and 2). Likewise I and J form an equivalence class for the possible parameters for R. Using this construct, we can represent 6 graphs using only 1 region. When the common region is large, this can be a substantial memory savings. Furthermore, this mechanisms allows us to generalize and create construct that we hadn't seen before. At the top of the figure, the AND node "P" makes it so that both Q and R are bound.

Thus, we have a mechanism for representing parameterized concepts in an Ontol

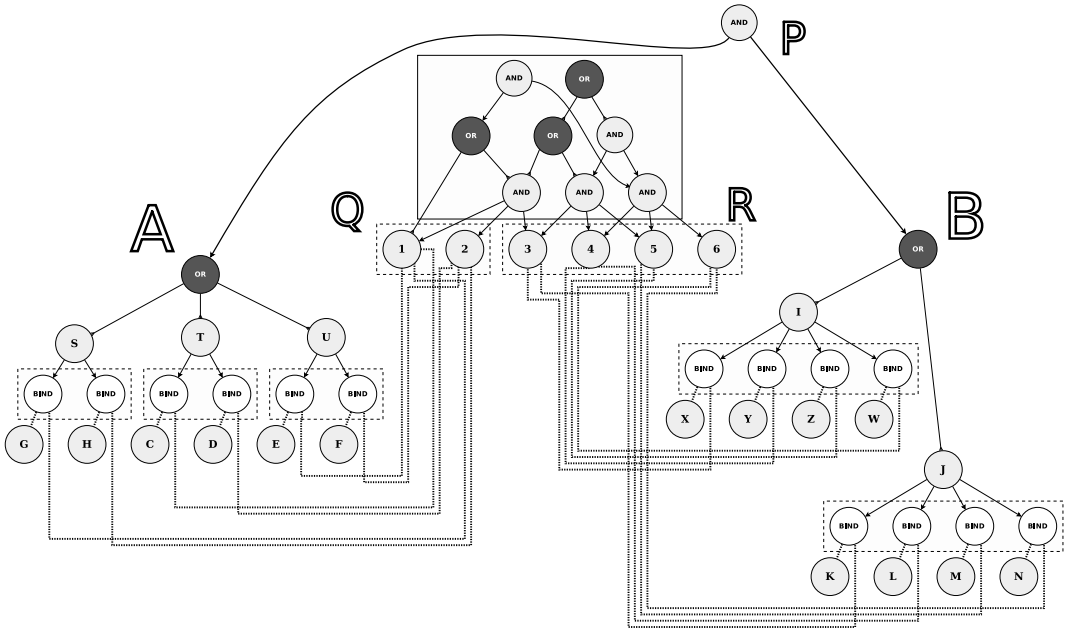


FIG. 4.4. **An example of a call graph with multiple parameterized input regions.**

framework. But the question remains for how these behaviorally isomorphic regions are found to begin with. For this, we introduce "behavioral signatures" in the next section, which allow us to quickly gauge the behavioral similarity among cortical regions.

4.3 Behavioral Signatures

In this section, we define what we mean by "behavioral similarity" of cortical regions. We also give an example algorithm as a proof of concept for "behavioral signatures" which allow us to quickly determine whether cortical regions are behavioral similar.

The behavioral signature we use is essentially a "hash", and uses the same basic idea that we use to encode invariant concepts in Chapter 3 (see Figure 2.8). That is, our signature is a bag of features that, though possible, it's unlikely that another cortical region will have the same set of features without without having the same behavior. Again, this hash is unlike a checksum in that we want similar behavior to have similar hashes. So the same principle that "binds" the dog-neck, dog-head, and dog-body can also be used to "bind" different behavioral features. That is, the dog's neck is basically a connection between the dog's head and the dog's body. It's the *overlap* that makes it so that we don't have to bind them explicitly.

We want our behavioral signatures to have the following properties:

1. If cortical regions A and B are behaviorally isomorphic, then the distance between their signatures should be small.

2. Vice versa. That is (assuming the parameters $sigTrials$ and $subsetSize$ from the algorithm in Table 4.1 are large enough) if 2 signatures are very close, the cortical regions should be behaviorally isomorphic.

3. If cortical region A is a sub-isomorphism of cortical region B, then A's signature will be a subset of B's signature.

For our behavioral signatures, we need to distinguish a set of nodes as "input" nodes. Because of top-down unfolding, input nodes are also output nodes as described by (Hawkins & Blakeslee 2004) and in Chapter 3. That is, we can set some "input" nodes to UNSPECIFIED and some to ON or OFF, then run our truth-derivation algorithm from Chapter 3, which will set the UNSPECIFIED nodes to ON or OFF. The other nodes are considered "internal" nodes. Because we're interested in behavior, not structure, we treat the internal nodes as a black box. The behavior of the cortical region, then, is essentially how the input nodes all affect each other. An important constraint for generating signatures is that they must be invariant with respect to input node ordering. This is because we're looking for regions that are behaviorally isomorphic, but we don't know the ordering on inputs.

As an example, suppose we want to generate the signature for the cortical region in Figure 3.2, with respect to the 10 input nodes 0 through 9. For this network, we might turn ON nodes 4, 5, and 7, turn OFF node 3, and leave nodes 0, 1, 2, 6, 8 and 9 UNSPECIFIED. When we run our truth-derivation algorithm from Chapter 3, nodes 6, 8 and 9 get turned ON as shown in Figure 4.5. (Some internal nodes get turned ON, but we're only interested in the external behavior of this cortical region.)

Once we have our set of input nodes, we generate the behavioral signature by

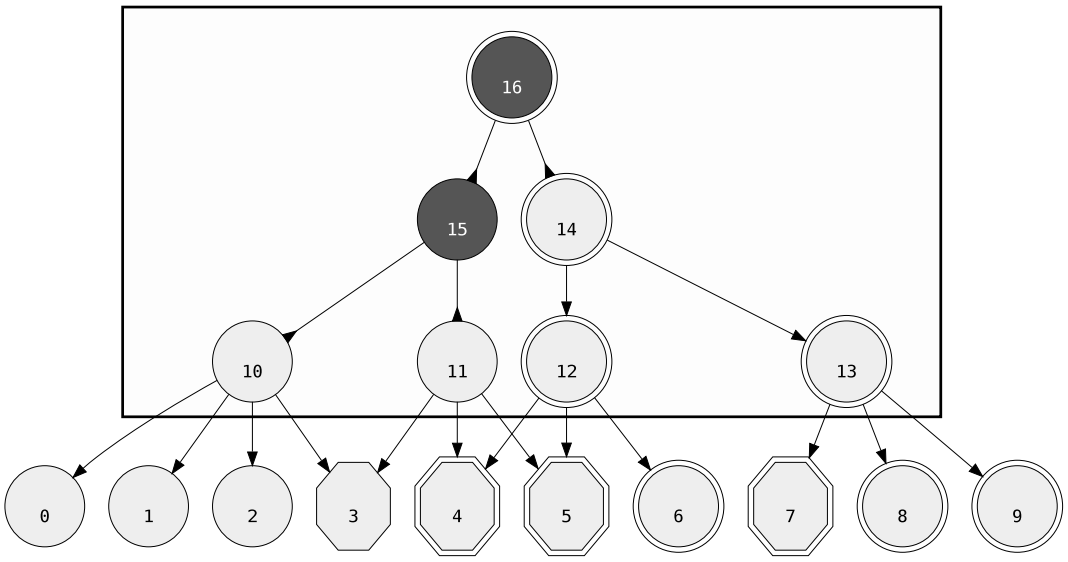


FIG. 4.5. **A call to a cortical region.** Nodes 0-9 are "input" nodes, while the remaining "internal" nodes are treated as a black box. Turning ON nodes 4, 5, and 7 and OFF node 3 causes nodes 6, 8, and 9 to turn ON.

randomly choosing subsets of nodes, and viewing the other nodes' truth-values as a result of changing the values of each subset. In our example, suppose we choose a subset of size 3, and suppose we want to sample 1000 times.

To do a single sample, we randomly grab 3 of the input nodes. For each of the 8 ON/OFF configurations, we set these 3 nodes, set the remaining 7 nodes to UNSPECIFIED, then run the truth-derivation algorithm from Chapter 3 to get an ON/OFF value for each of the 7 remaining nodes. For each of the remaining 7 nodes we create a "Karnaugh line", which is the value for this node under each of the 8 ON/OFF configurations. For example, if the Karnaugh line for node 1 in terms of nodes 2, 5, and 6 is `TTTFTFFT`, this is shorthand that node 1 is ON for 5 of the 8 permutations of for nodes 2, 5, and 6 being ON or OFF. The 1st case being that

these are all ON, etc..

These lines have been canonicalized. That is, all 6 orderings of the 3 nodes were generated and the Karnaugh line that came last alphabetically was chosen. For example, the line `TTFTFTFT` would be canonicalized to `TTTFTFFT` because the latter is generated by swapping the 1st and 2nd variable for the former. So we have only 96 entries, and not 256.

The pseudocode for finding the behavioral signature of a cortical area is given in Table 4.1. In this algorithm, the *canonicalizeKarnaugh* algorithm works by exhaustively trying all *subsetsize*! permutations for the inputs to a Karnaugh map, then reading the map from top to bottom, left to right, and returning the map with the smallest numeric value, if we treat $true$s as binary 1s, and $false$s as 0s.

For illustration, the full signature from Figure 4.5 is shown in Figure 4.6 (along with signatures for 3 other networks). These signatures were generated with 1000 samples, giving a total of 7000 Karnaugh line tallies per signature. The tally is given for each of the 96 Karnaugh lines. So the Karnaugh line `FFFFFFFF` occurred 1112 times for network **A**. These signatures give us what we're looking for: we're able to quickly tell whether 2 cortical regions are behaviorally isomorphic or not. Note that the Euclidean distance between the signatures for **A** and **B** is small compared to the distance between **A** and **C**. Note that these signatures are independent of the node ordering, and are dependent only on the behavior of the internal nodes, not on the structure. The distances shown in the bottom right of Figure 4.6 serve as proof-of-concept that we *can* create a measure of behavior as a feature-set, where the signature is independent of node ordering.

Table 4.1. **The algorithm to compute the behavioral signature of cortical area C.**

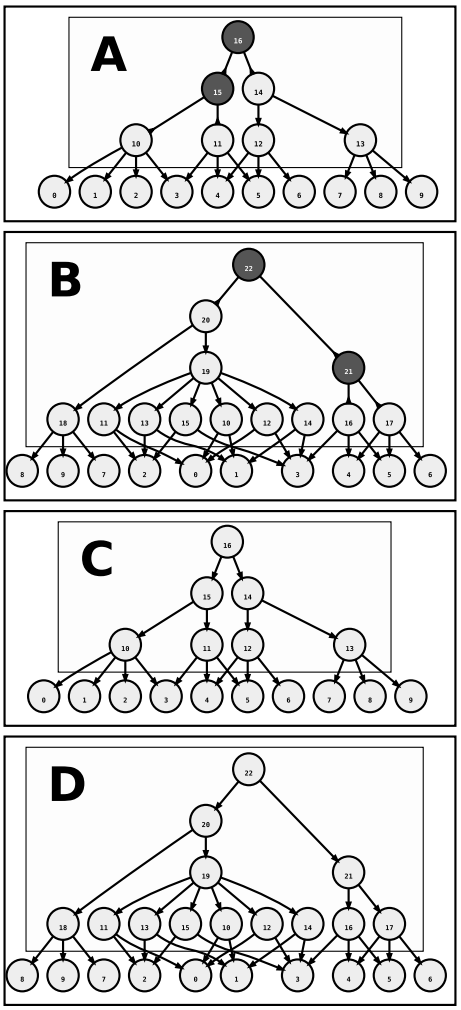
```
// C is a cortical region, I is the set of input nodes.
// samples is the number of random samples to take (in Figure 4.6 this is 1,000)
// subsetsize is the number of input nodes to force at a time (in Figure 4.6 this is 3)
// Returns a set of tallies, where each tally is for a canonicalized Karnaugh map with 2^subsetsize entries
define BehavioralSignature (C, I, samples, sigTrials, subsetsize)
  karnaughTallies = empty dictionary
  repeat sigTrials times
    sample = a random sample of subsetSize nodes from I
    // Initialize the karnaugh maps
    allMaps = empty dictionary
    // Now go through all 2^subsetSize values for these
    foreach ON/OFF truth value permutation p of the nodes in sample
      // Set the truth values
      set the nodes in C to their respective values in p
      run the parsing algorithm from 3.4 on C
      foreach node n in C − I
        // Add that node's value to the karnaugh map
        let allMaps[node][datuple] = n's truth value
    // Now canonicalize each map
    foreach map in allMaps
      canonicalizeKarnaugh (map)
      // increment the tally for map
      karnaughTallies[map]++
  return karnaughTallies
```

Also note that other isomorphism finding algorithms, such as the Structure Mapping Engine (Falkenhainer, Forbus, & Gentner 1989), SUBDUE (Holder, Cook, & Djoko 1994), and others will fail to see the similarity between **A** and **B** because these are different structures. The behavioral signatures introduced here are the only work known to us that will detect the behavioral similarity between these cortical regions.

4.4 Discussion

We have introduced mechanisms for integrating analogy into an Ontol framework and given proof-of-concept for these mechanisms, but there still remains much future work. For example, there's the question of how the cortical regions are segmented. There's the question of how behavioral signatures perform for networks with a large number of inputs, and whether we should restrict our segments to smaller cortical regions. Also, it may be fruitful to work directly in signature space for cortical areas. Since the behavioral signatures are *feature-sets* themselves, we should be able to run these signatures through (another instantation of) Ontol to efficiently represent relational concepts.

It should also be noted that by representing relational structures as static feature-sets, we're essentially addressing The Binding Problem (von der Malsburg 1999), which basically asks how feature-sets can be used to represent relational data in a neurologically plausible manner. This begs the question of whether our behavioral signatures or our BIND mechanism are neurologically plausible. We make no claims about the neurological plausibility of these mechanisms, but this might be worth investigation.



	A	B	C	D
FFFFFFFF	1112	1155	0	0
FFFFFFFT	383	380	0	129
FFTFFFFF	243	158	0	0
FFTFFFFT	190	151	0	35
FFTFFFTF	5	1	0	0
FFTFFFTT	2	13	0	3
FFTFTFFF	133	75	0	0
FFTFTFFT	346	363	75	42
FFTFTFTT	4	4	0	2
FFTTFFTT	0	2	0	1
FTTFFFFF	1	5	0	0
FTTFFFFT	25	11	0	12
FTTFFFTF	1	0	0	0
FTTFFFTT	1	5	0	0
FTTFTFFF	7	0	0	0
FTTFTFFT	10	35	121	185
FTTFTFTT	2	6	0	8
TFFFFFFF	470	259	0	0
TFFFFFFT	1679	1014	231	171
TFFFTFFF	110	52	0	0
TFFFTFFT	714	872	2007	2155
TFFFTTFT	0	0	0	4
TFTFFFFF	110	75	0	0
TFTFFFFT	346	372	102	70
TFTFFFTF	8	1	0	0
TFTFFFTT	20	19	0	0
TFTFTFFF	103	91	0	0
TFTFTFFT	734	1341	2172	1954
TFTFTFTF	3	4	0	0
TFTFTFTT	12	21	46	23
TFTFTTFT	0	0	0	4
TFTTTFTT	0	2	0	2
TTFFTFFF	0	5	0	0
TTFFTFFT	160	246	1370	1000
TTFFTTFT	0	0	0	2
TTTFFFFF	5	3	0	0
TTTFFFFT	16	37	126	153
TTTFFFTF	6	2	0	0
TTTFFFTT	8	22	58	25
TTTFTFFF	0	3	0	0
TTTFTFFT	27	153	596	839
TTTFTFTF	0	1	0	0
TTTFTFTT	4	41	96	174
TTTTTFTT	0	0	0	7

$\|A-B\|_2$	=	926,354	(9.72%)
$\|A-D\|_2$	=	9,109,628	(30.49%)
$\|A-C\|_2$	=	9,546,816	(31.21%)
$\|B-D\|_2$	=	5,544,956	(23.79%)
$\|B-C\|_2$	=	5,843,940	(24.42%)
$\|C-D\|_2$	=	301,794	(5.55%)

FIG. 4.6. **A demonstration of the utility of behavioral signatures for networks with similar behavior, but dissimilar structure, and vice versa.** Networks **A** and **B** are structurally different, but behaviorally similar, as are networks **C** and **D**. Networks **C** and **D** are structurally similar to **A** and **B**, respectively, but have ANDs where **A** and **B** have ORs, and are therefore behaviorally different, respectively. The columns at the top-right are the behavioral signatures for the 4 networks. The Euclidean distance (using the behavioral signatures) is shown for the 6 distances among the networks, with the percentage of the maximum possible distance between 2 cortical regions under these parameters.

Some of these ideas are discussed further in Subsection 8.2.3.

Chapter 5

DISCOVERY OF USEFUL MAPPINGS

When fully implemented, the parameterized mappings from Phase 2 should give us true transformation invariances, such as translation for visual data, that chunking and merging alone won't produce. We do this by creating equivalence classes from the parameters of a parameterized concept. But we still won't have generalized *relations* among feature-sets. That is, we won't have extracted the relations themselves. For example, we won't have a general idea of "rotating a shape by 45 degrees". To find these general relations, we propose searching for *mappings*, which transform a feature-set to another.

We propose that useful mappings can be found by finding mappings that allow us to further compress our data set. In this chapter, we provide the mechanics of mappings and we show how we find and use mappings to compress our data. We assume that we're given a set of conjunctions that we are told are equivalent. (This should be the output of Phase 2.) For example, we might be told that various rotations of the "dog" shape are the same image, as are various rotations of other shapes, as shown in Figure 5.1. Though we know these are all instances of the same invariant

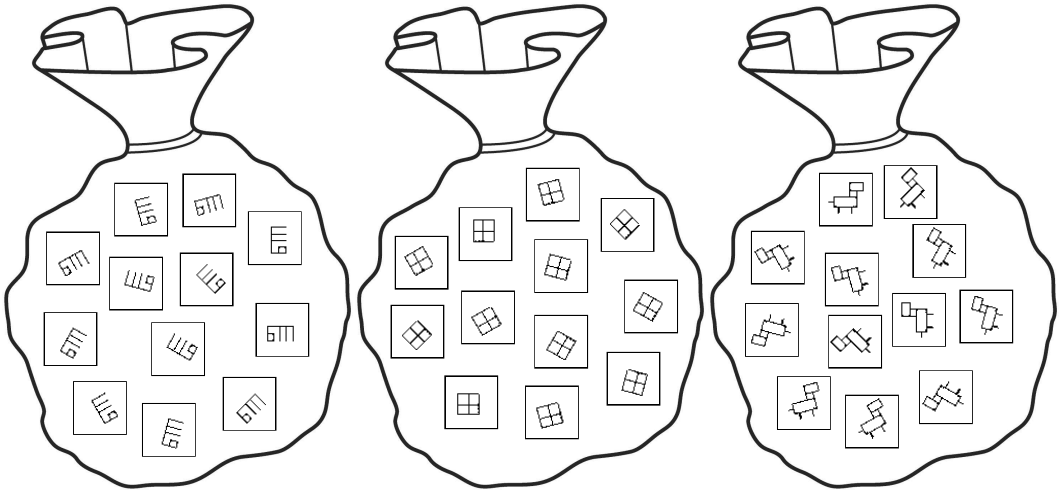


FIG. 5.1. **Bags of feature-sets that we know are all the same shape, though we don't know the relation among the instances in a bag.** Note that each image in each bag is actually a bag of features itself.

concept, we don't know the relation these feature-sets have with each other. Our task is to discover relations that are consistent across bags.

Apart from the general related work discussed in Section 1.3 and the analogy work discussed in Section 4.1, we know of no work that discovers mappings or makes use of these mappings to reduce description length. We believe that the work introduced in this chapter is novel.

5.1 The Problem of finding Mappings

In this section, we describe the problem of finding mappings. If we consider Figure 5.1 again, we note that each image is actually a bag of features itself. To further illustrate this, we've constructed a set of equivalence classes by hand that are

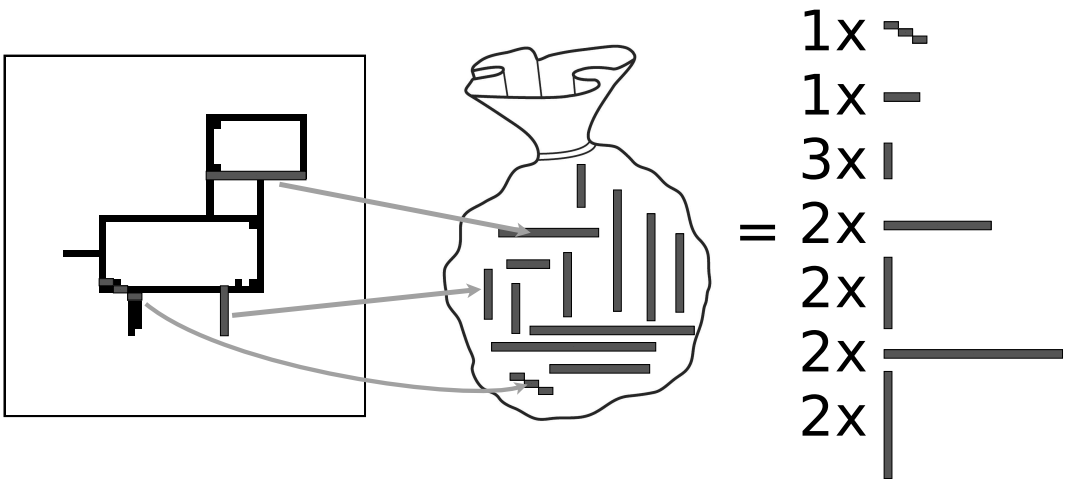


FIG. 5.2. **A translation invariant representation of the "dog" concept as a bag of features.** We simply tally the lines of every length and orientation. We also "bin" the lines by rounding the segments' length to the nearest multiple of 5. Note that under this transformation a shape rotated 180 degrees will have the same "signature" as the original shape.

invariant to translation and rotation. Namely, these features are the tallies of lines at different lengths and orientations as shown in Figure 5.2. Though we've hand coded these features, we assume that we will have discovered equivalence classes like these as a product of Phase 2.

Essentially, each "invariant" feature in Figure 5.2 is a large OR as illustrated in Figure 5.3. This shows how this type of feature might have come from Phase 2. That is, the output of Phase 2 is an ontology of ANDs and ORs. The ORs (or equivalences classes) that are invariant to translation (and other transforms) are found by discovering behaviorally isomorphic regions of the ontology. We assume that this problem has been solved in starting Phase 3.

Figure 5.4 shows this representation for our bag of "dog" instances. We have a

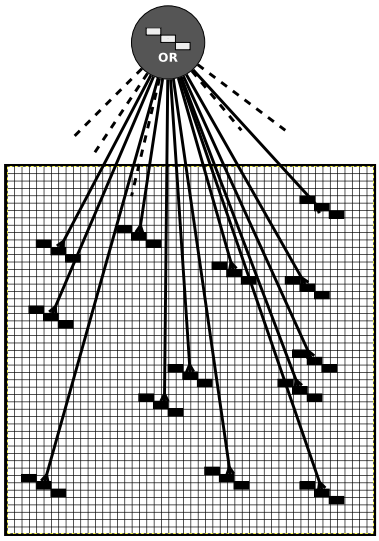


FIG. 5.3. **A translation invariant line segment as a large OR.** Each line segment shown on the grid is actually an AND pointing to the component pixels.

similar "vector bag" for each of our other 180 shapes.

Suppose we pick a pair of instances S_1 and S_2 from the bag of "dog" instances, and we want to find a mapping A such that $S_2 = A(S_1)$. For this example, let's suppose that S_1 is the "non-rotated" dog and that S_2 is the dog rotated 15° (shown at the right in Figure 5.4). To do this, we must first define how mappings work.

5.2 Mechanics of Mappings

A mapping is a set of ordered pairs, where each element of the pair is a feature (primitive or higher level). For example, the mapping A from Subsection 2.2.3 (in our overview of the the solution to building our "rickety bridge") is the set

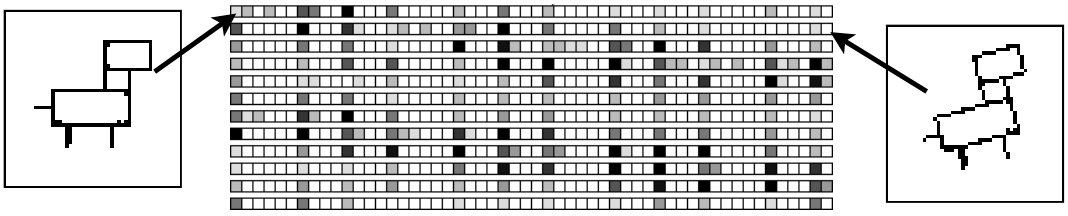


FIG. 5.4. **Our hand-coded invariant representation for rotations of the "dog" shape.** The top row is the feature count for the non-rotated dog shape shown on the left. The next row is the shape rotated 15° (shown at right), and so on to a rotation of 165°. Black means that the value for this feature is 10 or above (meaning that this feature occurred at least 10 times), and white means a feature value of 0.

$$\{ \quad (000.05, 045.05), \quad (000.10, 045.10), \quad (000.15, 045.15), \quad (000.25, 045.25),$$
$$(090.05, 135.05), \quad (090.10, 135.10), \quad (090.15, 135.15), \quad (090.25, 135.25) \quad \}$$

When we apply mapping A to a set S (denoted $A(S)$), we get a new set N, which is determined by the algorithm in Table 5.1. This algorithm creates a new set N by going over each item x of set S, if x matches the right side of a rule $r \in A$, then the left side of r is added to N. If there is no rule in A whose left side matches x, then x itself is added to N.

In each mapping, we assume that the 1st element of each pair is unique. Note that N and S aren't proper sets, but *bags*. That is, these data structures are unordered, but may have duplicated elements. For example, if we apply mapping A to the set $\{000.05, 000.15, 040.05, 045.05\}$, we get $\{045.05, 045.15, 040.05, 045.05\}$. Note that the element 045.05 is repeated twice.

Table 5.1. **The algorithm to compute the set ($A(S)$) resulting from applying the mapping A to set S.** Essentially, this creates a set N, which is the mapping A applied everywhere S has a key z_0 in A. If an element x has no key in A, then the new set simply contains the "unmapped" item x.

```
// We call this A(S) for short
define ApplyMapping(A, S)
  let N = {}
  foreach x ∈ S
    if ∃_{z∈A} s.t. x = z_0
    then let N = N ∪ {z_1}
    else let N = N ∪ {x}
  return N
```

5.3 What Makes a Useful Mapping?

Now that we've defined how mappings behave, we return to our example problem: finding a mapping A such that $A(S_1) = S_2$. Now, we can see that there are many possible mappings. For example, there are 10 features in S_2 that have a count of 4. (Due to anti-aliasing "noise", there are only 5 features in S_1 that have a count of 4.) This means that if a feature in S_1 maps to any one of these 10 features in S_2, it could just as equally map to any of the others. Therefore, we need a further constraint on what mapping we want. We need a metric that gives the utility for a mapping.

As we've been doing, we use the Minimum Description Length principle, and we define a mapping's utility as how much it allows us to compress our data set. This means it's never worthwhile to define a mapping over 2 instances of just one shape[1], because it will take more bits to define the mapping than will be saved by using it.

[1]Here, we use the term "shape" for clarity with our example. However there is nothing about this algorithm that's particular to images or shapes. The more general term would be "invariant class".

Table 5.2. **The algorithm to compute Description Length gain ($DLgain$).**

```
// The gain from using mapping A to describe feature bag Y using another feature bag Z.
// X is also a feature bag where X = A(Z).
define DLgain(X, Y)
  let gain = 0
  // We gain the cost of expressing Y.
  for k ∈ Y_s
    let gain = gain + Y(k)
  // But we have to pay for the differences between X and Y.
  // Here, we define Y(k) = 0 if k ∉ Y_s and X(k) = 0 if k ∉ X_s
  for k ∈ X_s ∪ Y_s
    let gain = gain − |Y(k) − X(k)|
  return gain
```

Instead, we must use the mapping for several shapes. For example, the mapping that essentially rotates a shape 45° is useful for all 3 shapes in Figure 5.1.

Given a mapping A, we define its utility for a pair of instances of a shape S_1, S_2 as the savings in description length by using $A(S_1)$ to express S_2. If no savings is gained, we simply use S_2 itself and we have a description length gain of 0.

If X is the result of applying mapping A to a feature bag Z, then we define the Description Length gain (or $DLgain$) by the algorithm in Table 5.2. Essentially, if we have Y "inherit" from X, we no longer have to express every item of Y. X might have some differences from Y, and these differences come at a cost. This is what the algorithm computes. Remember that a feature may occur multiple times. We use the notation $X(k)$ to be the number of times that feature k occurs in X. We also define X_s to be the *set* created from X, which has only 1 of each item from X.

This algorithm essentially assumes that Y has been "paid for" by X, then computes the expense (in terms of what must be added or deleted) of changing X to turn

it into Y.

Using $DLgain$, we define the utility U of using mapping A on shape instance S_1 to express shape instance S_2 as

$$U\left(A, S_1, S_2\right) = \max(0, DLgain\left(A\left(S_1\right), S_2\right))$$

We apply the mapping everywhere that doing so yields description length savings, so the total utility of our mapping A is this savings minus the cost to specify the mapping (which is twice its length since it specifies 2 items per entry). Formally, if C is our set of invariant classes (or shapes in our running example), this is

$$U\left(A\right) = -2\left|A\right| + \sum_{S \in C} \sum_{S_1, S_2 \in S \times S} U\left(A, S_1, S_2\right)$$

This utility function makes sense for the shapes domain. Note that all our mappings are multiples of 15°. Therefore, a mapping that's not a multiple of 15° shouldn't be that useful. Figure 5.5 shows the utility gains for mappings of various degrees.

5.4 Searching for Mappings

Now that we have our utility function for what makes a good mapping, we want to search over this space to maximize this function. This space is large, so a blind search might be inefficient.

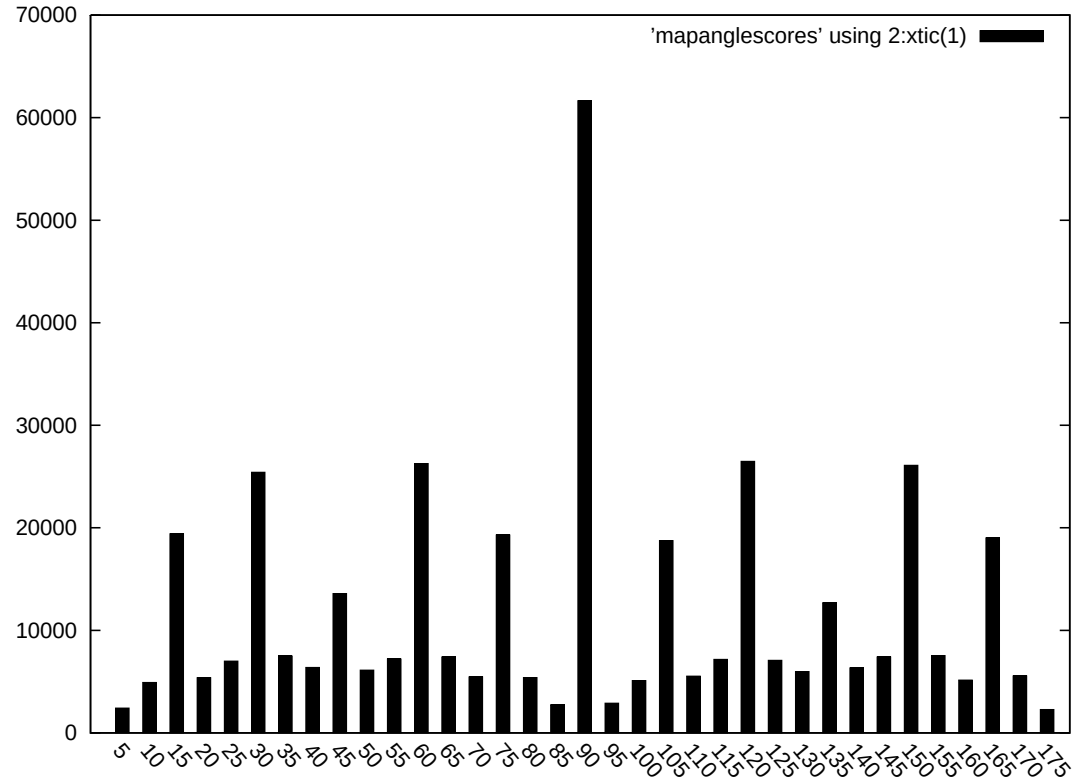


FIG. 5.5. **The scores (essentially description length savings) for hand-coded "rotation" mappings.** Since our data all are rotations of multiples of 15° of each other, rotations that aren't a multiple of 15° won't be as useful. These mappings aren't completely useless though because of anti-aliasing "noise". For example, the "Rotate 55°" mapping still places a lot of its rotations to around 60°, which is what these features may have been set to due to noise. The *Rotate*90° mapping has an especially high score because this rotation has almost no anti-aliasing noise.

Our search is doubly hard. Suppose we grab instances S_1 and S_2 from the bag of "dogs". Further suppose an oracle told us that the mapping from S_1 to S_2 is the same as the mapping from T_1 to T_2 and from R_1 to R_2, where T_1, T_2 are instances from a different bag and R_1 and R_2 are instances from yet another bag. At this point, it's still not trivial to find a mapping that best maps all these pairs. But our problem is even more difficult because we don't have an oracle telling us what shape T_1 maps to under the same transformation.

Our current algorithm seeks to maximize compression by searching over mappings in a 2 phase iterative process. Our first phase is finding a mapping given a *pairing* between instances of shapes. Starting with a randomly chosen shape S, we arbitrarily choose 2 instances of that shape S_1 and S_2. For these 2 instance, we do a "best match" among the features. We do this by constructing a bipartite graph and then doing bipartite graph minimization. To construct the graph, each "node" represents a feature in S_1 or S_2, where node $N_{i,j}$ corresponds to feature j of S_i. If S_1 and S_2 have a different number of features (as they do for our representations in Figure 5.4), then we "pad" the smaller feature-set with new features with frequency values of 0. We create an edge between every pair of nodes $N_{1,i}$ and $N_{2,j}$ where the value of this edge is the difference in the frequencies of the respective features. That is $E_{i,j} = |S_{1,i} - S_{2,j}|$. We then run a standard bipartite graph matching algorithm to *minimize* the sum of the values over the chosen edges. This matching generates a mapping, where we map from feature i to j if edge $E_{i,j}$ has been chosen. Then we enter our second phase, which is adjusting our pairings among the instances of our shapes. Starting with this "seed" mapping M, for every other shape T, we search over all pairs of instances in T to get T_1 and T_2, where $DLgain\left(M\left(T_1\right), T_2\right)$ is maximized. Using this pairing for the rest of the shapes, we find another matching by

constructing another bipartite graph. This time, the value of each edge is given as $E_{i,j} = \sum_{R \in shapes} |R_{1,i} - S_{R,j}|$. We then construct a new mapping using the bipartite matching minimizing algorithm as described as before. Using the new matching, we readjust our pairing and we repeat until convergence or until a maximum number of iterations is exceeded.

We ran this search algorithm on our rotated object data. There were 181 shape types at 12 orientations each, for a total of 2,172 shapes. Our algorithm was told what shape type each instance was, but not the orientation (which is the output we'd hope to get from Phase 2). Using this algorithm, we successfully discovered the 12 transformations *and* the pairs of instances for each shape type for which these mappings apply.

Figure 5.6 shows the comparison of these discovered mappings to our hand-coded mappings from Figure 5.5. The mappings discovered by our system were competitive with the hand-coded mappings in terms of description length savings, outperforming the hand-coded mappings in all instances but 2. Note that the "between" angle rotations (such as 5°) were never discovered by our algorithm because these rotations didn't yield any further compression on our dataset once we have the multiple-of-15° rotations. This is because the differences of angle in the instances of a shape in our data were always multiples of 15°. It's possible that the hand-coded mappings have too much overhead. For example, they encode how to rotate a line of length 30, when lines of these lengths are rare in our images.

For illustration, part of the discovered 90° mapping is shown in Table 5.3. For the most part, this matches with our hand-coded mapping. For example, this mapping converts a line of length 5 at orientation 0° to a line of length 5 at orientation 90°.

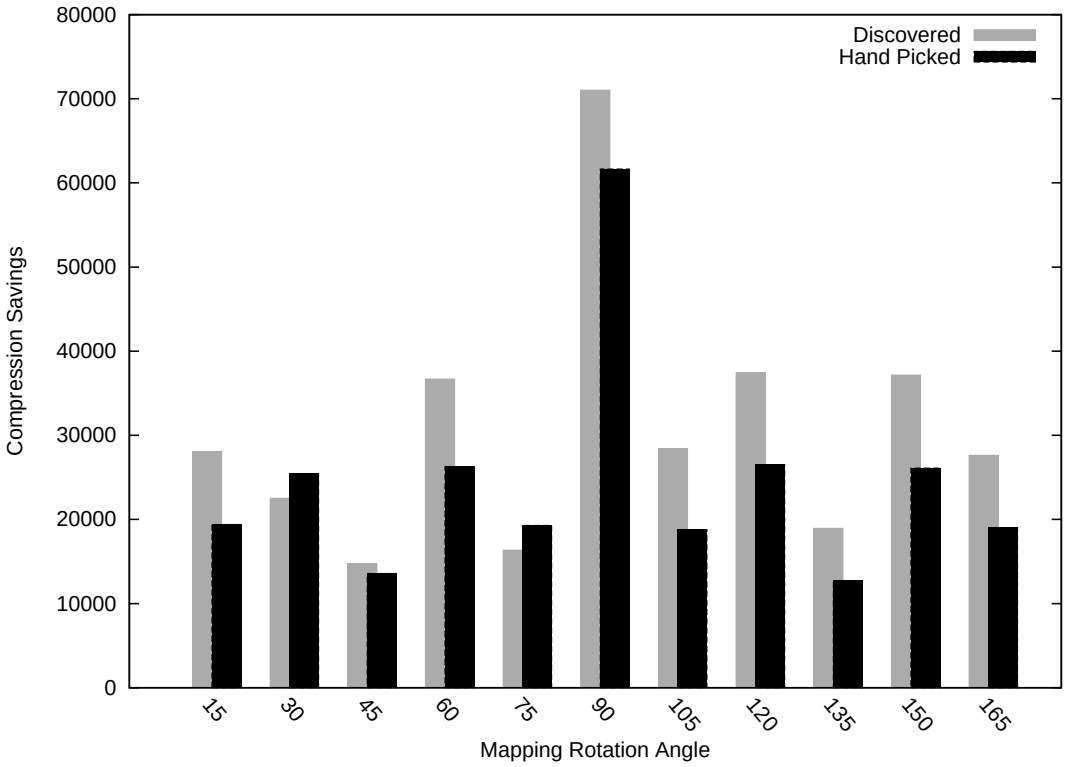


FIG. 5.6. **Comparison of hand-coded mappings vs. mappings found by our algorithm.** The description-length savings is shown for using each of these on our 181 images. The "Hand Coded" data points are taken from Figure 5.5.

Table 5.3. **Part of the 90° mapping discovered by our algorithm.**

000.05	→	090.05	005.05	→	095.05
000.10	→	090.10	005.10	→	095.10
000.15	→	090.15	005.15	→	095.15
010.05	→	100.05	015.05	→	105.05
010.10	→	100.10	015.10	→	110.10
010.15	→	150.15	015.15	→	035.10
045.05	→	135.05	175.05	→	085.05
045.10	→	135.15	175.10	→	085.10
045.15	→	060.30	175.15	→	085.15

Our search had some mistakes. For example, 010.15 should have been mapped to 100.15, not 150.15. We suspect that this is because of the relative rarity of lines of length 15.

For illustration, the table below shows the results of a 45° "rotation" for the "dog" shape from Figure 5.4. The leftmost column is the list of features. The 2nd column is the respective feature values for the original shape (shown at the left of Figure 5.4). The 3rd column is the features values for the dog shape rotated 45°. Also shown are these feature values resulting from applying our hand-coded and discovered mappings for a 45° rotation shown with along with the mapping and the error for each feature in terms of number off from the actual rotated shape. The total sum of squared error is shown in the final row, with our discovered mapping having a slightly lower total error than the hand-coded rotation.

Feature	Original (Rot. 0°)	Actual Rot. 45°	Hand-coded Map	Val	Err	Discovered Map	Val	Err
000.05	1	2	→ 045.05	2	0	→ 045.05	2	0
000.10	2	0	→ 045.10	0	0	→ 045.10	0	0
000.20	2	0	→ 045.20	0	0	→ 045.20	0	0
005.05	5	2	→ 050.05	2	0	→ 040.05	2	0
005.10	4	0	→ 050.10	0	0	→ 065.05	0	0
010.05	13	5	→ 055.05	1	+4	→ 035.05	2	+3
015.05	4	5	→ 060.05	3	+2	→ 025.05	3	+2
020.05	2	2	→ 065.05	2	0	→ 070.05	2	0
025.05	4	12	→ 070.05	3	+9	→ 050.10	4	+8
030.05	2	15	→ 075.05	5	+10	→ 075.05	5	+10
035.05	1	12	→ 080.05	12	0	→ 095.05	13	-1
035.10	0	1	→ 080.10	0	+1	→ 080.10	0	+1
040.05	1	5	→ 085.05	5	0	→ 085.05	5	0
040.10	0	2	→ 085.10	4	-2	→ 170.10	1	+1
040.15	0	2	→ 085.15	0	+2	→ 040.15	0	+2
045.05	1	1	→ 090.05	1	0	→ 090.05	1	0
045.10	0	2	→ 090.10	2	0	→ 085.15	2	0
045.20	0	2	→ 090.20	2	0	→ 155.10	2	0
050.05	2	5	→ 095.05	5	0	→ 080.05	4	+1
050.10	0	1	→ 095.10	4	-3	→ 000.10	4	-3
050.15	0	2	→ 095.15	0	+2	→ 030.15	0	+2
055.05	1	9	→ 100.05	13	-4	→ 055.10	12	-3
055.10	0	2	→ 100.10	0	+2	→ 055.10	1	+1
060.05	1	17	→ 105.05	4	+13	→ 105.05	5	+12
065.05	1	11	→ 110.05	2	+9	→ 130.10	4	+7
070.05	4	3	→ 115.05	4	-1	→ 110.05	2	+1
075.05	8	2	→ 120.05	2	0	→ 115.05	2	0
080.05	8	1	→ 125.05	1	0	→ 145.05	2	-1
085.05	8	1	→ 130.05	1	0	→ 140.05	1	0
090.05	4	1	→ 135.05	1	0	→ 135.05	1	0
090.10	1	0	→ 135.10	0	0	→ 140.10	0	0
090.15	2	0	→ 135.15	0	0	→ 135.10	0	0
095.05	6	1	→ 140.05	2	-1	→ 150.05	1	0
095.10	2	0	→ 140.10	0	0	→ 130.05	0	0
100.05	8	1	→ 145.05	1	0	→ 120.05	3	-2
105.05	4	3	→ 150.05	1	+2	→ 155.05	1	+2
110.05	2	2	→ 155.05	1	+1	→ 160.05	4	-2
115.05	3	3	→ 160.05	4	-1	→ 100.05	8	-5
120.05	3	8	→ 165.05	8	0	→ 165.05	8	0
125.05	2	6	→ 170.05	8	-2	→ 005.05	0	+6
130.05	2	7	→ 175.05	8	-1	→ 010.05	2	+5
130.10	0	0	→ 175.10	0	0	→ 015.10	1	-1
135.05	2	4	→ 000.05	4	0	→ 000.05	4	0
135.10	0	0	→ 000.10	1	-1	→ 000.15	2	-2
135.15	0	0	→ 000.15	2	-2	→ 135.15	0	0
140.05	2	8	→ 005.05	6	+2	→ 175.05	8	0
140.10	0	0	→ 005.10	2	-2	→ 125.15	1	-1
145.05	1	6	→ 010.05	8	-2	→ 040.10	8	-2
150.05	3	6	→ 015.05	4	+2	→ 170.05	6	0
155.05	2	5	→ 020.05	2	+3	→ 020.05	4	+1
160.05	3	1	→ 025.05	3	-2	→ 015.05	2	-1
165.05	5	2	→ 030.05	3	-1	→ 060.05	3	-1
170.05	12	3	→ 035.05	2	+1	→ 055.05	3	0
175.05	5	1	→ 040.05	2	-1	→ 030.05	2	-1
175.10	4	0	→ 040.10	0	0	→ 050.05	0	0
	$\sum Err^2$				543			515

5.5 Discussion

The process of using mappings is another mechanism for compressing feature-set data. Although we've used the process here on output from Phase 2, we can use it anywhere we have sets of feature-sets that are known to be equivalent. Furthermore, the mappings we found that best compress our data for a rotated image domain are

those that correspond to rotations, which is what we assume a person would use for describing this domain. So we argue that compression is a good metric for finding mappings that correspond to what a person might use.

Chapter 6

APPLES TO ANGLES AND THE MACGLASHAN TRANSFORM

This chapter outlines the final phases of the bridge, but a full investigation of the ideas in this chapter are the subject of future work.

As discussed in Subsection 2.2.4, we'd like to characterize the mappings that we learned in Phase 3. Recall in Figure 2.12 that we want to generate "graphs" for the behavior of mappings. Essentially, we want to have mechanisms that allow us to discover and exploit the fact that part of our theory of angles is isomorphic to part of our theory of apples. The process of generating graphs for the behavior of mappings, then finding and exploiting isomorphisms in these graphs is Phase 4. To do Phase 4, we must answer the following Questions:

1. How do we get the behavior of our mappings?
2. How do we represent the mappings' behaviors as graphs?
3. How do we segment these graphs to give graphs for different *types* of mappings?

(For example, how do we separate Rotation transforms from Translation transforms, since they both operate on the same type of data?)

4. How do we find isomorphisms among the behavioral graphs for different mappings?

5. How do we use these isomorphisms once we find them to compress our data (and to generalize and to transfer knowledge)?

Phase 5 is the process of representing graphs, particularly behavioral graphs, as feature-sets, then using Ontol to create an ontology of these graphs on which we can do recognition and inference. The process of turning a graph into a set of features is called "The MacGlashan Transform".

6.1 Related Work

For the general idea of characterizing behavior of mappings for a system, we know of little related work aside from the work on analogy that we reviewed in 4.1.

For the MacGlashan Transform, the idea of representing large graphs as feature-sets is not novel. For example, (Shervashidze *et al.* 2009) and (Przulj 2007) both represent graphs by sampling small connected sub-graphs called "graphlets", putting the graphlets into a canonical form (at exponential cost in the size of the graphlet, which is usually small to make this feasible), then representing the graph as a distribution over the graphlets. This distribution is called the "kernel" of a graph. Using these kernels, they are able to quickly estimate whether two large graphs are isomorphic. This distribution is essentially a feature-set, and we borrow heavily from these

ideas. However, no one we know of has taken the next step: creating an ontology of graphs by chunking and merging these feature-set representations of graphs.

6.2 Building and Using "Behavioral" Structures

In our running example, the intuition behind the answer to Question 1 —how we get the behavior of our mappings— is that we want to "play around" with rotations to generate a structure of relations between rotations. We learn *rules*, such as that if we apply Rotate5 to a feature-set, then apply Rotate40, we get roughly the same result as if we applied Rotate45.

The basic idea is that our nodes are "tagged" sets of features (where each item has a unique tag), and our edges are mappings. These feature-sets come from the mappings themselves. That is, the feature-sets will be all the features to which a mapping applies, or the *domain* of the mapping. For example, suppose we have the 3 mappings below (where Mappings A, B, and C correspond to 30°, 60°, and 90° rotations, respectively):

Mapping A	**Mapping** B	**Mapping** C
000.05 → 030.05	000.05 → 060.05	000.05 → 090.05
000.25 → 030.25	000.25 → 060.25	000.25 → 090.25
030.05 → 060.05	030.05 → 090.05	030.05 → 120.05
030.25 → 060.25	030.25 → 090.25	030.25 → 120.25
060.05 → 090.05	060.05 → 120.05	060.05 → 150.05
060.25 → 090.25	060.25 → 120.25	060.25 → 150.25
090.05 → 120.05	090.05 → 150.05	090.05 → 000.05
090.25 → 120.25	090.25 → 150.25	090.25 → 000.25
120.05 → 150.05	120.05 → 000.05	120.05 → 030.05
120.25 → 150.25	120.25 → 000.25	120.25 → 030.25
150.05 → 000.05	150.05 → 030.05	150.05 → 060.05
150.25 → 000.25	150.25 → 030.25	150.25 → 060.25

If we start with Mapping A, then our tagged feature-set will be a set of attribute-value pairs, where the first item of each pair will be from the *domain* of Mapping A, and the value will be a gensym created by attaching an "o" (for "original value") to the first item. So, our initial tagged feature set will be the following attribute-value pairs:

$$
\begin{aligned}
Pairs\,(A) = \quad \{ \quad & \texttt{000.05} = \text{o.}\texttt{000.05}, \quad \texttt{000.10} = \text{o.}\texttt{000.10}, \\
& \texttt{030.05} = \text{o.}\texttt{030.05}, \quad \texttt{030.10} = \text{o.}\texttt{030.10}, \\
& \texttt{060.05} = \text{o.}\texttt{060.05}, \quad \texttt{060.10} = \text{o.}\texttt{060.10}, \\
& \texttt{090.05} = \text{o.}\texttt{090.05}, \quad \texttt{090.10} = \text{o.}\texttt{090.10}, \\
& \texttt{120.05} = \text{o.}\texttt{120.05}, \quad \texttt{120.10} = \text{o.}\texttt{120.10}, \\
& \texttt{150.05} = \text{o.}\texttt{150.05}, \quad \texttt{150.10} = \text{o.}\texttt{150.10} \quad \}
\end{aligned}
$$

Note that our tags are the original values for each feature. This allows us to keep track of what becomes of each feature. Furthermore, if o.`000.05` is the original count for the number of times `000.05` appeared in a feature-set, then we can use this for

calculating the number of times a feature occurs in a mapped feature-set. When we apply mapping A to this tagged set, we get the tagged set

$$A\left(Pairs\left(A\right)\right) = \begin{array}{ll} \{ & \texttt{000.05} = \text{o.}\texttt{150.05}, \quad \texttt{000.10} = \text{o.}\texttt{150.10}, \\ & \texttt{030.05} = \text{o.}\texttt{000.05}, \quad \texttt{030.10} = \text{o.}\texttt{000.10}, \\ & \texttt{060.05} = \text{o.}\texttt{030.05}, \quad \texttt{060.10} = \text{o.}\texttt{030.10}, \\ & \texttt{090.05} = \text{o.}\texttt{060.05}, \quad \texttt{090.10} = \text{o.}\texttt{060.10}, \\ & \texttt{120.05} = \text{o.}\texttt{090.05}, \quad \texttt{120.10} = \text{o.}\texttt{090.10}, \\ & \texttt{150.05} = \text{o.}\texttt{120.05}, \quad \texttt{150.10} = \text{o.}\texttt{120.10} \quad \} \end{array}$$

If we apply Mapping B on the same initial set, the resultant feature-set will be

$$B\left(Pairs\left(A\right)\right) = \begin{array}{ll} \{ & \texttt{000.05} = \text{o.}\texttt{120.05}, \quad \texttt{000.10} = \text{o.}\texttt{120.10}, \\ & \texttt{030.05} = \text{o.}\texttt{150.05}, \quad \texttt{030.10} = \text{o.}\texttt{150.10}, \\ & \texttt{060.05} = \text{o.}\texttt{000.05}, \quad \texttt{060.10} = \text{o.}\texttt{000.10}, \\ & \texttt{090.05} = \text{o.}\texttt{030.05}, \quad \texttt{090.10} = \text{o.}\texttt{030.10}, \\ & \texttt{120.05} = \text{o.}\texttt{060.05}, \quad \texttt{120.10} = \text{o.}\texttt{060.10}, \\ & \texttt{150.05} = \text{o.}\texttt{090.05}, \quad \texttt{150.10} = \text{o.}\texttt{090.10} \quad \} \end{array}$$

So our graph for applying these mappings will be as shown in Figure 6.2. This Figure also shows how we might answer Question 2: how we represent the mappings' behaviors as graphs. In this figure, each node represents a particular setting of attribute-value pairs, the edges represent our 3 transformations: A, B, and C.

We can make some inferences from this graph. For example, we see that mapping C is its own inverse. We see that mapping A applied twice is the same as applying mapping B once. We might want our algorithm to infer that A is a "more primitive" mapping than B or C because multiple applications of A can yield the same result as a single application of B or C, but not vice versa. Thus, A might become a candidate for a primitive mapping operation.

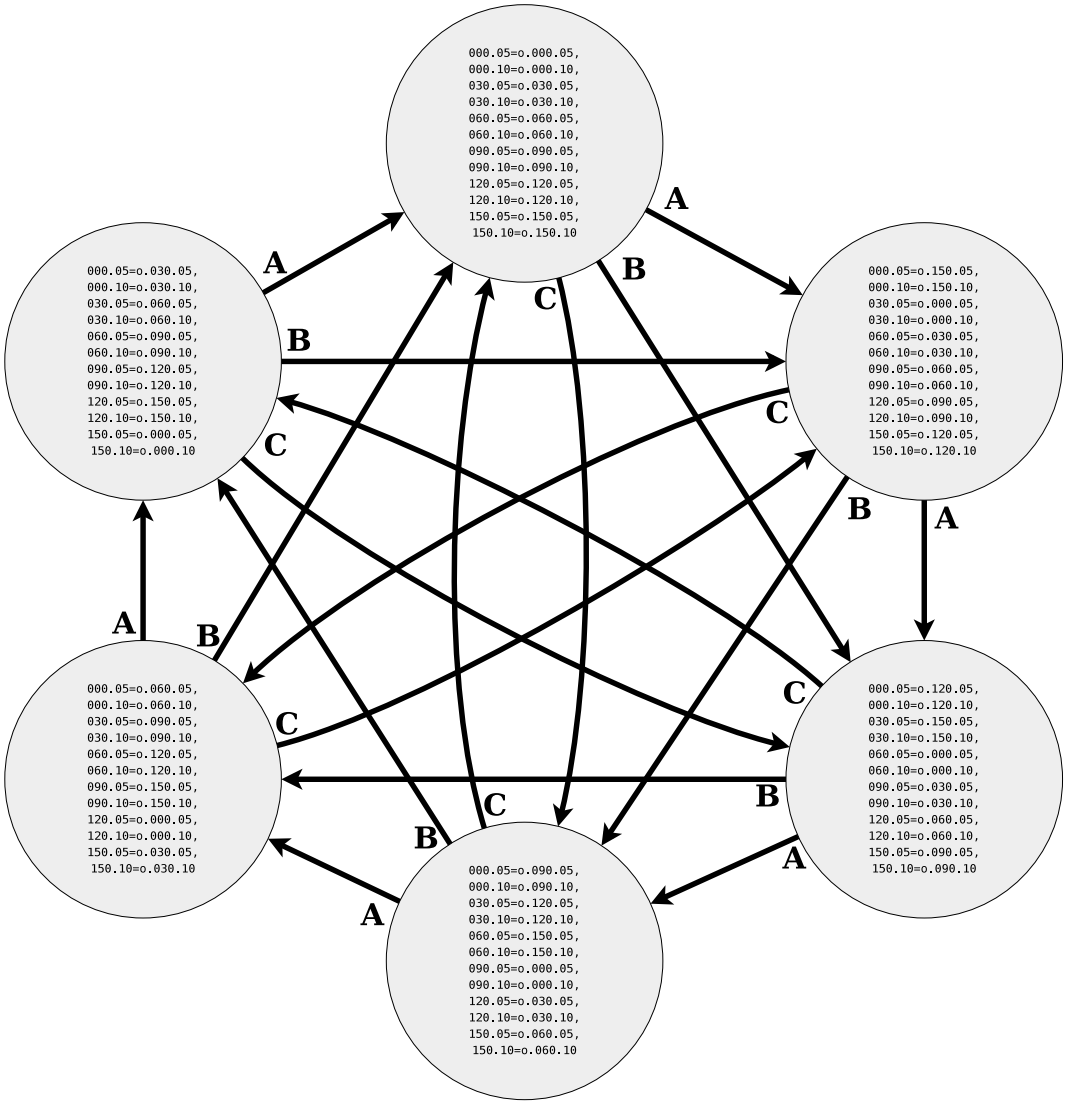


FIG. 6.1. **Behavior of mappings represented as structure.** We show 3 mappings A, B, and C corresponding to rotation by 30°, 60°, and 90°, respectively. The nodes correspond to the settings for the attribute-value pairs for features on which these operations operate. The top node corresponds to the tagged feature-set that we constructed to get $Pairs\,(A)$. In a real example, this would be only part of the graph. We'd also have other mappings such as translation and scaling, in addition to other rotations.

The graph in Figure 6.2 is exhaustive for the 3 mappings, but on a real example, we'll likely have many more mappings, and the domains and ranges of these mappings won't necessarily be the same. So in real life examples, we won't always afford to be exhaustive. Thus, the graphs need to be created by "sampling" the behavior of the mappings. How to do this sampling —for example, randomly vs. based on input data— is an open question which we leave to future work.

In a real life example, the graph in Figure 6.2 would be connected to a much larger graph, so we need to address Question 3, how we segment these graphs. One approach would be to apply the SUBDUE system (Holder, Cook, & Djoko 1994), which finds frequently occurring subgraphs in a larger graph. If successful, this would also address Question 4, how we find isomorphisms in the "behavior of mappings" graph. Applying a SUBDUE-like algorithm is also the subject of future work.

This leaves Question 5, which asks how we might use these isomorphisms once we find them. The answer to this falls into 2 main categories:

Compression As in Phase 2, if we see that we have mappings that are essentially the same (or have isomorphic components), then we can represent these commonalities *once* instead of having to repeat them each time.

Generalization and Transfer of knowledge For example, if we develop a gisty theory of number, then we can apply this knowledge when we see new data. An example of this would be learning about distances. Just a few samples will suggest that distances behave like other quantities, and we can then test this hypothesis. Then we can quickly make inferences about distances without having to first redevelop a theory of number for distances.

6.3 The MacGlashan Transform: Representing Relational Structures as Feature-Sets

The MacGlashan Transform is left to future work, but here we describe the possibilities if we're able to represent relational structures as feature-sets.

The feature-set graphlet kernels of (Shervashidze *et al.* 2009) and (Przulj 2007) both show that this representation of graphs is possible. Conceivably we can store our relational structures in "kernel" space. That is, we can use Ontol on these feature-sets to create an ontology of graphs for efficient storage, and for doing inference on graphs. We can also create a "Reverse" Transform, which converts feature-sets back to relational structures. This might be a constraint satisfaction search problem: given a hash, search for what structure produced the hash.

Representing graphs as feature-sets essentially uses the same trick that we use in representing "dog" as a collection of ANDs and ORs: namely, the use of *overlap* to do binding. The concept of a dog *is* a relational structure. We have dog-neck which is a relation between dog-body and dog-head. We never explicitly state that a particular dog-head is the *same* head that the dog-neck binds to, but it's hard to find a different head that fits these constraints.

The benefit of having this transform is that it allows us to represent a huge variety of concepts as feature-sets because such as large variety of knowledge can be represented as graphs. Then, if we have an efficient Ontol system, we can use this system to do learning and inference on almost any kind of knowledge.

6.4 Meta-Cognition: Feeding the Dragon its Tail

There's a lot to be said about Meta-Cognition, so we won't say much here. With just a few added mechanisms, we propose that we might be able to have some metacognitive properties, such as having our search algorithm develop its own heuristics. The overview for how this might be done is this:

1. We *represent* the search processes itself as feature-sets.
2. We use Ontol to *chunk and merge* these feature-sets. This should generate common search patterns and find common waypoints in searches.
3. Then we *apply* the patterns Ontol finds to our new searches.

If our feature-set representation is truly general, then we can express our working memory (for searches) in the same language as we do our regular memory.

Ideally, this process will generate "macros" analogous to the lemmas that humans develop while proving theorems in geometry. For example, imagine if a person were to run Conway's Game of Life by hand. Doubtless, a person would create all sorts of cognitive shortcuts. For example, if a "glider" got to an open spot, the person would probably not bother doing the calculation for each step of the glider, but would rather correctly predict that the glider would be $\frac{t}{4}$ cells to the right and $\frac{t}{4}$ cells down in t time steps. (With today's algorithms, the computer is blindly simulating the glider, it'd have no conception of what a glider is.) Ideally, we'd want the computer to recognize these computational shortcuts also.

We've only scratched the surface of this idea. A fuller investigation will be the subject of future work.

6.5 Discussion

We've shown a proof of concept of how relational theories might be represented as feature-sets, and how we might create a general gisty theory of "quantity", and finally complete our bridge from raw sensor data to such a rich theory. There is a good deal of overlap between Phase 4 and Phase 2. Both Phase 4 and Phase 2 *extract and parameterize* similarities in behavioral. Ideally, we'd like to unify much of Phase 4 and Phase 2, and have Phase 4 be just a few added mechanisms such as generation of "behavioral" data and representing the data as feature-sets.

There is yet much work to be done on this bridge, and on Artificial Intelligence in general. We discuss this future work in the remaining chapters of this dissertation.

Chapter 7

WHAT'S LEFT FOR AI?

> Let an ultraintelligent machine be defined as a machine that can far surpass all the intellectual activities of any any man however clever. Since the design of machines is one of these intellectual activities, an ultraintelligent machine could design even better machines; there would then unquestionably be an "intelligence explosion", and the intelligence of man would be left far behind. Thus the first ultraintelligent machine is the last invention that man need ever make, provided that the machine is docile enough to tell us how to keep it under control. It is more probable than not that, within the twentieth century, an ultraintelligent machine will be built and that it will be the last invention that man need make.
>
> –I. J. Good (Good 1965)

Solving The AI Problem —creating a machine that's unquestionably "intelligent" on the same level or above a human being— would be a mark in human history to rival the advent of agriculture, writing, printing, and powered flight. In this chapter, we discuss a set of open problems for Artificial Intelligence and how we can extend

the work discussed in this dissertation to achieve this goal.

7.1 Open Problems for AI

In (Pickett, Miner, & Oates 2007) and (Pickett, Miner, & Oates 2008) we discuss a set of open problems for Artificial Intelligence. These are problems for which we believe even conceptual solutions are currently weak. We list some of these problems here.

These problems are meant to be intuition pumps. They are meant to point out shortcomings of current techniques in AI, and stimulate algorithms for how these problems might be addressed. These particular problems were chosen because they're feats that people can do easily, but for which we know of no work that says how a computer can solve them even in principle. We hypothesize that progress on these problems will significantly advance the state of research on Artificial General Intelligence.

7.1.1 The 3D Pen Problem

In his Allegory of the Cave (Plato 360 BC), Plato describes a group of people whose observations of the world are solely shadows that they see on the wall of a cave. The question may arise as to whether these observations are enough to propose a theory of 3-dimensional objects. In principle, this problem can be solved. If an agent is given a representational framework that's expressive enough to encode a theory of 3-dimensional objects, then the agent could go through the combinatorially huge

number of theories expressed in this language (under a certain length) and choose the one that best explained the data (where "best" can be defined in terms of Ockham's Razor (or The Speed Prior, described in Subsection 8.2.1)). The best theory will likely include a description of 3-dimensional objects (assuming such a theory is of unrivaled utility for characterizing the data). Thus, our task is possible, given an exponential amount of time. There are other real examples of building theories of phenomena that aren't directly observable: neither atoms, genes, radio waves, black holes, nor multi-million year evolutionary processes are directly observable, yet scientists have built theories of these.

A robot given sonar sensor data (or uninformed visual data) is faced with fundamentally the same problem. Visual observations of a 3-dimensional object, such as a pen, can be very different (in terms of raw sensor data) depending on whether the pen is viewed lengthwise or head on. Therefore, the ability to propose "scientific theories" of this type is something a cognitive architecture should be able to explain. The architecture's representation framework needs to be expressive enough to encode such theories, and the architecture's model-builder should be able to discover theories in polynomial time.

A similar approach can be used to create causal theories from observational data (Pearl 2000). Statisticians point out that it's impossible to prove causality from observational data. This is true, but we can use our "theory language" to propose causal theories that are more likely than others.

We view this as being fundamentally the same problem as developing a theory of Euclidean Geometry (given various theorems, posit a small set of axioms that produce the axioms), molecular chemistry (given various interactions, posit the existence of

atoms (and how they work)), or basic genetics (given plant phenotypes, posit the existence of genes).

7.1.2 Green Glasses

When a person dons a pair of green-tinted sunglasses for the first time, they have little trouble adapting to their altered visual input, but this isn't such a trivial task for a (visual) robot. In terms of raw sensor data, a green-tinted scene has very different values from the same scene in its natural color. We suspect that this is because people have abstract representations that are invariant of the instances that caused them. Representations developed from visual data should also be invariant to translation, rotation, and scaling. These *invariant representations* aren't limited to visual data. A stenographer can hear different speakers say the same phrase in different pitches, volumes, and speeds, yet produce the same transcription.

7.1.3 RISK

We claim that there exist concepts that people form naturally, yet are not handled by existing concept discovery systems. For example, consider the strategy game "RISK". The map in Figure 7.1 shows 6 Continents, each of which are subdivided into areas, which we'll refer to as Territories. The goal of this game is to dominate the map by controlling all the Territories. At any point in the game, each Territory is controlled by only one player, and a player can gain control of other Territories by invading from neighboring Territories that they already control. Every Territory that a player controls has one or more Armies stationed there. The odds of an

invasion being successful increase with the number of Armies stationed in the invading Territory, and decease with the number of Armies in the defending Territory. Players receive a Continent Bonus (in the form of extra Armies at the beginning of their turn) for controlling all the Territories of a Continent.

When people play RISK, they generally develop the concept of what might be called a "Border Escalation". An example of this concept would be what often happens when one player controls the Continent of North America, and another player controls South America. In this case, both players place many Armies on their respective "Border Territories" (namely Central America and Venezuela) while keeping few Armies on the "Interior Territories". Placing Armies on the Border Territories strengthens a player's defense against invasion (and therefore helps ensure that player's Continent Bonus), but also acts as a threat to the other player's Continent Bonus. Therefore, the other player adds more Armies to their Border Territory, and the process escalates. An analogous similar process is also often seen in between Siam and Indonesia. Thus, the *relationship* between Siam and Indonesia is analogous to the relationship between Central America and Venezuela, even though Indonesia and Venezuela are almost as different as two Territories can be in RISK. Note that when people play RISK, this concept is *unsegmented*. That is, no one separates out the Border Escalation that occurs between Siam and Indonesia; it's integrated with the rest of the game-play. People readily form other concepts while playing RISK. For example, the general idea of a "border" and expansion of controlled Territories, a "cascade" where one player's dominance will propel itself, and the case where one player tries to weaken another by invading only a single Territory in the other's Continent. These concepts allow us to concisely characterize games of RISK. Most of these concepts are still useful for describing games played on an alternate map (e.g.,

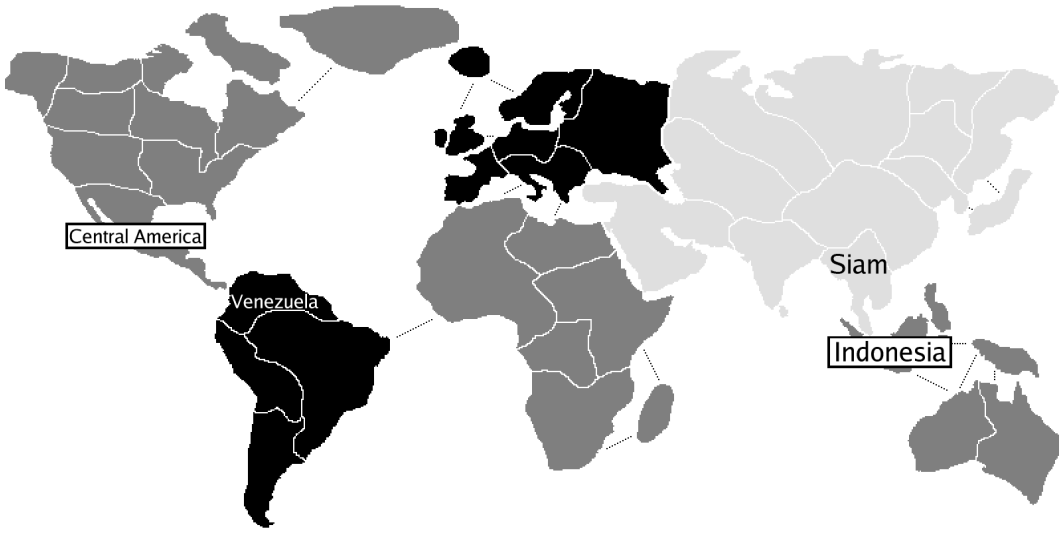


FIG. 7.1. **"Border Zone": A concept formed by analogy in RISK.** Siam is to Indonesia as Central America is to Venezuela. The Continents are shown as contiguous regions of same-colored Territories.

of a hypothetical world with different Territories and Continents). This generality is useful for transfer of learning. We argue that many of these concepts are difficult even to represent with existing approaches, much less discover. Note that one need not take actions to form these concepts. They can be formed merely by observing others playing RISK, and this unsupervised approach is what we will be focused on.

7.1.4 The Smashed Banana Problem

In search, there is also "The Smashed Banana Problem". Suppose we're cooking supper for people, and we're thinking about how to make the noodles not stick together. We're searching through possible scenarios, and possible actions we can do. One possible action we could take is to smash a banana against our forehead. We

would guess that this branch is never even brought up to be evaluated (to see if we should search it) since even if it only takes a millionth of a second to discard, there are still billions of branches. Of the possibly infinite number of actions we could do, how do only the "most sensible" ones come to mind?

One alley to look for a solution to The Smashed Banana Problem is connectionism. This is where associations are built betwixt various semantic areas. One need only follow connections to have a finite number of possible branches on which to search. There is also hierarchical connectionism (or frames and rule inheritance).

The basic idea behind hierarchical connectionism is that there are rules over a general class of objects/phenomena/whatever and that these rules are inherited by members of the class. For example, suppose you want to flatten a piece of wood. You can realize that wood is in the class of hard materials. Hard materials require forceful actions. Forceful actions usually require hard materials, and, more specifically, usually tools. Now there are only so many tools that work with wood, and there are only so many possibilities to consider.

7.1.5 Specifying Reward: The Tomato Harvester

> "And how will you inquire into a thing when you are wholly ignorant of what it is? Even if you happen to bump right into it, how will you know it is the thing you didn't know?"
>
> –Meno's Paradox, from Menon III of Pharsalus (Plato 387 BC)

An agent should be able to plan and take actions to attain a (possibly externally specified) goal, and an architecture will have to address *how* goals are specified. This

is non-trivial for a robot baby because it's born with a minimal model of the world and therefore, no "language" to express what should cause a reward. Furthermore, we shouldn't rely on any particular sensor modality to specify the reward. That is, the specification of the reward should be invariant to the raw sensor data.

Human brains seem to have solved this problem. For example, the majority of male humans seem to be innately attracted to women, and vice versa. From a computational standpoint, telling the difference between a man and a women is far from trivial. The attraction seems to be invariant to any single modality: most people either blind or deaf from birth still follow this pattern.

An approach to solving the problem is given in the following example: Suppose we wanted a robot's innate goal to be to harvest tomatoes. Furthermore, we wanted the robot to harvest only proper tomatoes (ripe, but not over-ripe, not too small, or insect infested, etc.), and we don't know what its sensor suite will be. (In practice, we might "teach" the robot as we would with a human, but this example is for illustrative purposes.)

To do this, we could build several robots with widely ranging modalities, and have them (over several months) experience the environment of the tomato fields. From this experience, the robots should have built a world model that would include a "tomato taxonomy". Then, we can find an invariant representation of a "good tomato" by noting what parts of their ontologies are "active" when they have experiences with good and bad tomatoes (analogous to what some neuroscientists do to localize various cognitive functions in humans in fMRI studies). Specifically, we could find an invariant representation for the act of harvesting good tomatoes. We can then put this in an agent's "innate" model, and specify that it's a goal. Then,

when developing representations, the agent should discover a representation that is similar (or perhaps isomorphic to) the representation of this goal.

7.1.6 Story Summarization

People seem to be good at summarizing stories. For example, suppose Tim watches an animated movie called "The Boy Who Wanted to be a Bear". With seemingly little effort, he might summarize the story like this:

> The story takes place in Alaska or Northern Canada, wherever polar bears and Eskimos live. There's a polar bear couple who loses their cub, so they go into the human village and kidnap a human baby boy. They then raise the baby. When the baby grows up, he decides he wants to be a bear. Then some magic stuff happens, and the boy becomes a bear.

A few things to note about this summary:

- Tim summarized the story with seemingly little effort.
- Tim never went down to the pixel level. He never said what color hair the baby had.
- There are a few specific events (such as going into the village and kidnapping the baby), and there are much broader events (such as raising the baby).
- Tim never explicitly mentioned that the bears raised the baby as a bear (and not a human), but we can only imagine that this was the case given that the

bears were trying to replace their cub (and that the boy ended up wanted to be a bear).

We suspect some of this will be doable if we can represent relational data as feature-sets and then throwing the feature-sets into a system like Ontol. If we can represent stories as a collection of relations (facts), then we can extract out what facts are common among multiple stories, so we might be able to come up with abstract relational concepts such as "raising a baby". We suspect Ontol will have to use its mechanism for parameter binding. This binding will probably be lazy. For example, we might not initially think about the manner in which the bears raised the baby initially, so we might bind this later, after we deduced it.

If we pump out a behavioral signature for Tim's summary of the boy who wanted to be a bear, this should be similar to the signature for the "pixel-level" story. (Or the same way that the higher-level node representing an invariant "dog" summarizes the actual pixel-level dog even though information is lost.)

7.1.7 Emergent Phenomena: Traffic Jams and Conway Gliders

Emergent phenomena are of particular difficulty for existing concept formation algorithms. For example, a particularly tricky problem is to discover a representation of a traffic wave. That is, given a bird's eye view of a simulation of automobile traffic on a highway, a person can readily point out where the traffic jams are. A traffic jam is different from a collection of cars. Individual cars move in and out of a traffic jam, and the jam itself usually moves opposite the direction of traffic. A person could also create features to describe the traffic waves: e.g., its spread and how fast it's moving.

```
Time
  1    ....4....4...10.1.....3....4......5....1000......3
  2    ...4....4...30.1..2.......4.....5.....4000.1......
  3    .......4...300...2...3.........5.....5000.1.1.....
  4    ..........300.1.....3....4..........5000.1.1..2...
  5    ..........00.1.1.......3.....4......000.1.1..2...3
  6    ...4......00..1..2.........4......5.00.1.1..2...3.
  7    ..4.....5.0.1...2...3...........5..10.1.1..2...3..
  8    .4.....50..1..2....3....4.........20.1.1..2...3...
  9    4.....500....2..2......4....4.....0.1.1.1....3....
 10    .....500.1.....2...3.......4.....5.1.1.1..2.....3.
 11    ..4..00.1..2......3...3.........5.1.1.1..2...3....
 12    ....20.1..2...3......3....4......1.1.1..2...3....4
```

FIG. 7.2. **The traffic simulation.** Each number represents the speed of a car (moving to the right), and each dot (`.`) represents an empty space in the road (which loops around from the right to the left). For example, at time `1`, the car represented by a `3` at the right has a speed of 3, and is the same car that is represented by a `4` (since it accelerated) at the 4th spot from the left at time `2`. A car accelerates 1 spot per time step per time step until it reaches its maximum speed of 5, or until it would run into a car in front of it, at which point it reduces its speed to 0. In these 12 steps of the traffic simulation, we can see two distinct traffic jams, which individual cars move into and out of.

One of the simpler systems that exhibits emergent phenomena is the traffic model of Nagel and Schreckenberg (Nagel & Schreckenberg 1995). In this model, one of the first emergent concepts one notices is traffic jams. An example of this model is shown in Figure 7.2. In this model (a one dimensional cellular automaton), cars are represented by numbers which indicate their speed. At each time step (going from top to bottom), a car moves its speed to the right unless that would mean colliding with a car in front of it, in which case the car moves as many spaces as possible, then reduces its speed to 0. If after moving there are no cars in front of it, the car increments its speed by 1, up to a maximum of 5. The cellular automaton is circular such that if a car moves off the right end, it continues to the left end.

Note that this model is fully observable and deterministic, and the underlying

dynamics are also known. This is in contrast to many real systems, such as automobile traffic in Los Angeles, where the underlying dynamics are not known completely. For the deterministic model, forming concepts gives us no additional predictive power if computation time is not a factor. Thus we conclude that one of the benefits of forming concepts is characterization of a system at a lower computational cost. This is support for The Speed Prior, which we describe in Subsection 8.2.1.

In the traffic example in Figure 7.2, there are two traffic jams. These traffic jams have several characteristics that are not present in any of the cars or "snapshots" of all the cars. For example, although traffic is moving to the right, the traffic jams tend to move to the left. Also, the traffic jams have properties such as their size and their duration. Using the concept of a traffic jam, we can describe the system succinctly in these terms: "The system has two traffic jams approximately evenly spaced. The leftmost traffic jam has a width of about 2 or 3 cars, a speed of -1 (i.e. 1 in the leftwards direction), and a duration of about a dozen time steps..."

Another emergent phenomenon that we'd like our computer to recognize is a glider in Conway's Life, as we discussed in Subsection 6.4. A glider is like a traffic jam in that the cells that are "part of" the glider constantly change.

Likewise, a "dog" at 2 different translations is like a glider or a traffic jam in that the raw pixels that are ON when the dog is ON can change.

7.1.8 Conway's *Blurry* Life

Suppose our computer is shown many example runs of Conway's Game of Life. We'd want the computer to come up with the concept of a "glider", "beehive", and

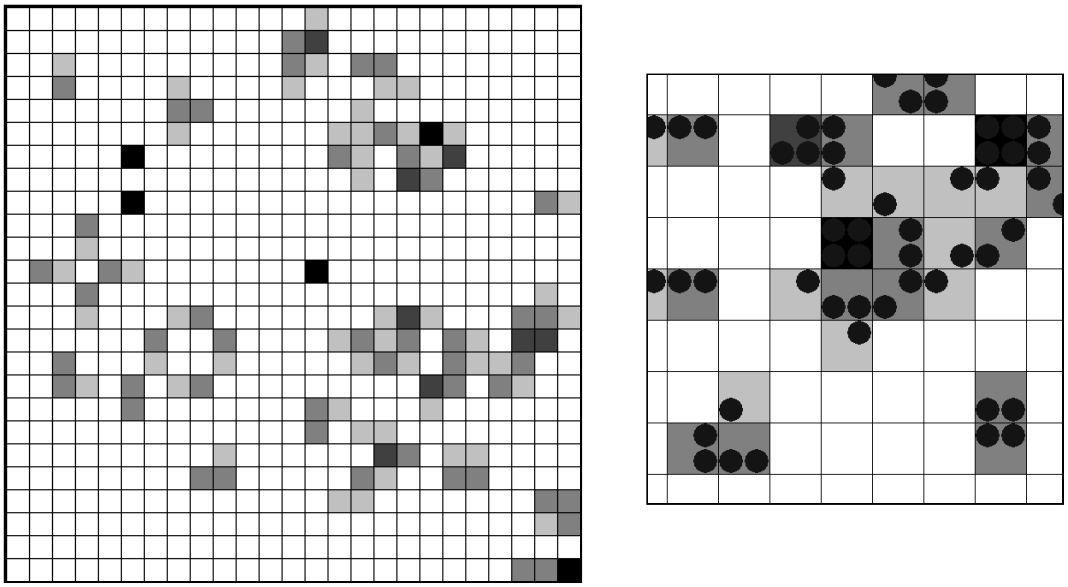


FIG. 7.3. **"Blurry Life".** On the left is Blurry Life as the computer sees it, where each pixel can take one of 5 values. The figure on the right shows the "internals" of blurry life. Each pixel is really 4 cells in Conway's Life, and the value of the pixel is the count of how many Life cells are "alive". Both figures feature a glider on their lower left. Note that on the right figure, the same Life cell configuration can result in different pixel values. For example, the "box" configuration of 4 alive life cells can be a single black pixel, 2 "50%" grey pixels, or 4 "25%" grey pixels.

other configurations that people name, but we'd also like our computer to eventually discover the "rules" of Life, for example that an OFF or "dead" pixel with exactly 3 ON neighbors becomes ON or "alive" the next time step. This is already a tricky problem, but we can make it even trickier by "blurring" the pixels by reducing our resolution so that every pixel is now a 2 by 2 "summary" of 4 cells, and is one of 5 values representing the number of alive cells in it. An example of "Blurry Life" is shown in Figure 7.3.

In Blurry Life, we'd like our computer to do the equivalent of coming up with

molecular theory. We'd like our computer to posit the existence of cells though they aren't directly observable. This problem is theoretically solvable using the exponential time algorithm given in Assumption VI. It's our hope and belief that there are faster algorithms for forming these theories. It's unclear whether the system described in this dissertation would be able to come up with such a theory or if we need additional mechanisms, such as postulating "theoretical entities", as discussed by Drescher (Drescher 1991). We hope that the ontology that's generated by Ontol will make the search for the theory more feasible by effectively reducing the dimensionality of the problem.

If we can solve this "Blurry Life" problem, it will pave the way for building models for which we don't have a simulation. For example, the work of (Berry *et al.* 2003) builds social simulations. Here, we're given data about the effects of social dynamics, but the underlying dynamics are unknown. For this domain, we need to go both "down" to create the model and "up" to find the macro behavior.

7.1.9 Simulation Speedups and Automatic Multi-scale Models

As mentioned in Section 6.4, a person running Conway's Game of Life by hand will likely develop shortcuts. If a person had the time and patience to calculate weather simulations by hand, that person would doubtless create shortcuts for this too. The amount of computation that goes into these simulations is so vast that no person could do much of a simulation, but a computer could. Conceivably, using the ideas generated in this dissertation, a computer could discover shortcuts and "lemmas" for simulating the weather and other complex models.

If we're able able to speed-up a simulation of Conway's Life to get a fast (though possibly lossy) model, we can apply these same techniques to a virtually unlimited range of simulations.

We can also apply these techniques to a virtually unlimited range of modalities: As we have an intuition for vision and sound, imagine a computer having an intuition for other modalities such as infrared, weather data (mentioned in (Hawkins & Blakeslee 2004)), or speed-sensor data. A good deal of a dolphin's cortex is dedicated to the processing of sonar. The sonar that's in submarines is a mere toy compared to what dolphins sense. Likewise, it's conceivable that a computer could develop this kind of intuition given a large amount of sonar data.

Since our system doesn't know how many dimensions we have, we should be able to apply our system to voxel data (MRI) and geographical and weather data for which people have bad intuitions. Conceivably, a computer could have an intuition about 3 or 4 dimensions the same way we have intuition about 2 dimensions.

When we model electrical systems, we can model on the level of individual electrons using Maxwell's equations, or we can model many electrons using Ohm's law.

Philosophical Aside: Super Bowl from Traffic Sensors We can simulate traffic from a global perspective. For example, if we have speed sensor data, then we can simulate things like Friday evening rush hour, traffic flow. Hypotheticals (e.g., if there was a blockage at a particular place on I-95). Such a simulation is shown in Figure 1.1.

There's the question of how deep a model of the world we can get from this

rather narrow stream of data. Suppose that somehow the world was relatively stable for an indefinite period of time, even trillions of years. Is it possible to develop a theory of the Super Bowl given nothing but trillions of years of traffic data? What if we use the exponential-time algorithm from Assumption VI? It's possible that there are only a finite number of possible universes, and the one that fits the traffic data also has the Super Bowl.

7.1.10 Theoretical Terms

We've seen that we can model traffic from a "global" perspective. We can also do traffic simulations from the perspective of an individual driver. For example, when we're driving, we'll note my local surroundings (including the view in the rear-view mirror). We'll note that one driver is "aggressive", while another driver is talking on her cell-phone. Based on these superficial features, we'll be able to make some prediction about how the local traffic will behave. We'll also be able to decide how to drive to avoid being locked in, get where we're going faster, etc.. For example, if a "slow" driver is in the left lane talking on his cellular phone, and an aggressive driver is coming from behind, we can predict that the fast driver will pass the slow driver on the slow driver's right.

7.2 Cognitive and Philosophical Answers

The ideas in this dissertation might also give us some leverage and "thought tools" to help answer some philosophical questions.

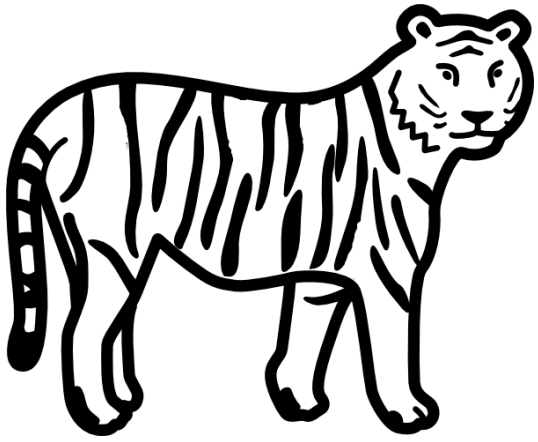


FIG. 7.4. **A tiger with 28 stripes.**

7.2.1 Tiger Stripe Problem

If people are briefly shown Figure 7.4, then asked to describe it, most of us would say it's a tiger. Most of us would be able to answer whether the tiger was in color or black in white, a photo or a line drawing, but not all of us would be able to describe which way the tiger was facing (as not all of us can say which way Lincoln is facing on the U.S. one cent coin, despite seeing thousands of pennies), and very few people would be able to accurately answer how many stripes the tiger had. Though most people have some *gist* of the number. They would say it's more than 3, but less than 1000. Ontol shows how a computer might learn a representation that would allow it to also remember that it saw a tiger without explicitly representing the number of stripes it has (though having a "gist" of the number). Ontol also explains why this is a useful feature of cognition.

7.2.2 "Tip of The Tongue" Phenomenon

The "tip of the tongue" phenomenon is where a person "knows" a piece of information, such as the name of a hotel, but can't seem to recall it. The person is able to *verify* the name (if asked whether another name is it), but seems unable to retrieve it from memory. One explanation for this from the perspective of Ontol is that it might be that we have the high-level signature or hash for the concept, but not the low-level sensors that generate the signature. Therefore, we're able to verify the name when we see it, but we can't "retrieve" the name directly. We instead resort to *search* to find low-level sensors that satisfy the *constraint* that the high-level node must be triggered.

7.2.3 Language

The idea of signatures might also shed light on some longstanding philosophical problems.

There's the question of Platonic ideals (Plato 360 BC). Whether there exists such a thing as an ideal dog, for example, and how this concept is learned. A hypothesis from the point of Ontol is that we are given a number of instances of "dog", then we discover a signature for all of them, then we posit an "ideal" dog that produces that signature most cleanly. If we have a formal theory (based on information theory) of how these signatures are generated, then we might be able to say what the ideal "dog" is be for a given world-history of an agent.

Language generation can also be seen as constraint satisfaction search: we have

some concept represented in Ontol's AND/OR ontology (we call this the "mentalese" described by (Fodor 1975) and (Pinker 1994)) and we search for a grammatical structure that produces this signature. Just as we learn to "draw" by searching for a construct of pixels that makes our invariant "dog" concept turn on, language is also constraint satisfaction search.

In both drawing and language construction, we learn a set of macros to speed up search. In drawing, we learn lines, basic shapes, and basic shading techniques. In language generation, we learn words, turns of phrase, and formulas.

Chapter 8

CONCLUSIONS AND FUTURE WORK

> Finally, to whatever degree of simplicity I may at last have reduced the Analytical Engine, the course through which I arrived at it was the most tangled and perplexed which probably ever occupied the human mind.
>
> –Charles Babbage (Babbage 1864)

In this dissertation we've provided the blueprints for a rickety bridge from raw sensor data to rich relational theories. Although we've provided a framework for solving some longstanding problems in Artificial Intelligence, there are still many holes to fill in before we fully solve these problems. In this chapter, we review what we've built so far, we explain what's left to do for building our bridge, and we propose a research plan for working toward a machine with human level intelligence.

8.1 Substantiation of Claims

In this section we review our claims from Section 1.4, and we show how these are substantiated.

Our first claim –providing a conceptual story for going from raw sensor data to rich relational theories– has been substantiated in Chapter 2, with supporting details in Chapters 3, 4, 5, and 6. We've provided a story, though rickety, for how we can get parameterized concepts from non-relational data, how we can find and use mappings among those concepts, and how we can find structure in the behavior of these mappings to get an intuitive theory of parameters such as scalars. Our bridge fulfils many of our desiderata given in Subsection 1.2.1: The use of Ontol as the backbone for our system makes for algorithmic elegance, where a single mechanism does the bulk of the work. We've shown the domain independence of our approach by demonstrating its parts on a wide range of domains. There is nothing in our approach specific to any single modality, and none of the parts are dependent on our own domain knowledge. None of the mechanisms require an inordinate amount of computational resources, and we've demonstrated this with several mechanisms by implementing and running them. Certain parts of our theory, namely parts of Ontol, are based on neurological models, and are therefore biologically plausible.

The *model* built by our system also fulfils the desiderata we set forth in Subsection 1.2.2: compression has been the guiding metric for each link of the bridge. As demonstrated in Section 3.6, the theory builds concepts that are useful for learning human-labeled data. We have not rigorously demonstrated that our system makes accurate predictions, but our model is based on that of (Hawkins & Blakeslee 2004),

which makes such claims about their model. We argue that, in future work, we can show that our model fulfils this desideratum. Using the same "top-down" mechanism as (Hawkins & Blakeslee 2004), we can make predictions. For example, in Figure 3.3, the best parse is y. When y fires, it turns ON w (a is already ON). w then fires and turns on b. So given that a, c, and d are ON, but missing the value for b, we accurately predict that b will be ON.

We've fulfilled our major technical claim, which is Ontol. As demonstrated in Chapter 3, ontol autonomously creates a useful set of concepts using a solid basis in Bayesian probability theory. We've shown how Ontol finds and uses these concepts, and we have demonstrated the utility of these concepts by showing how they can be used to yield significantly better compression on feature-set data than by using Lempel-Ziv alone.

We've also demonstrated the utility of these concepts by introducing a semi-supervised learning algorithm, based on Ontol, that outperforms the best (that we know of) existing semi-supervised learning algorithm for learning from a handful of positive examples. Our algorithm also gives us some insights on how people might be able to learn from just a handful of examples.

We've further shown the utility of the concepts learned by Ontol by an algorithm that uses Ontol to learn macro-actions in Reinforcement Learning. We've shown that this algorithm outperforms existing algorithm for learning macro-actions in Markov Decision Processes.

We'd like to point out that Ontol was designed with neither Semisupervised Learning nor Reinforcement Learning nor compression as the end-goal. The end-goal

was to build something that would help us span the chasm between sensor data and a rich theory of the world. This is where our goal remains, though showing progress towards *this* goal is not as straightforward as it is for these side-problems.

We've also fulfilled our 3rd technical claim: Equation 3.1, gives us a theoretically sound measure for evaluating the utility of ontologies, and provides theoretical backing for future work in unsupervised learning.

We've fulfilled claim 4 in Section 4.2, where we detail how parameterized concepts are represented in the same representation framework as Ontol.

In Section 4.3, we introduced and implemented *behavioral signatures*, which give us an efficient way to search for behavioral similarity among parts of the ontology built by Ontol. In Figure 4.6, we show, as a proof of concept, how these signatures can be used to quickly gauge the behavioral similarity among different regions of the conceptual structure. This substantiates claim 5.

Claim 6 is substantiated in Chapter 5, in which we show how mappings among concepts are represented, discovered, and used to compress our model.

8.2 Future Work: Taking a Stab at Solving The AI Problem

> If what you are doing is not important, and if you don't think it is going to lead to something important, why are you at Bell Labs working on it?
>
> –Richard Hamming (Talk at Bellcore, March 7th, 1986)

Though we've made a basic outline for how a computer can learn rich relational theories from raw sensor data, there's still much work to be done, not only to fill in this bridge, but also to develop a fully intelligent system. We hypothesize that building our bridge will give us insights into the nature of intelligence and will help us find cognitive mechanisms that will help us solve some of the problems described in Section 7.1.

To work toward Artificial Intelligence, we propose the following methodology:

1. Create a list of phenomena of general intelligence, for example as we've done in (Pickett, Miner, & Oates 2008). Also create a list of problems that people easily solve, but that computers have a hard time with, for example the problems introduced in Section 7.1.

2. Create a representation framework, a set of mechanisms that operate on this framework, and an architecture for how these mechanisms interact.

3. Tell a story about how these mechanisms address the list of phenomena and solve the set of problems.

4. Implement these, test them on diverse domains, and see what is and isn't addressed.

5. Modify the set of mechanisms (and goto step 2).

The general solution for building our bridge is to keeping "plugging holes" in our system by formalizing, implementing, and testing the parts. We also need to work out the details of how all these parts will work together. Our general plan is to plug all the holes in Ontol, learn relational structures as described in Phases 2-4, represent

relational structures as feature-sets as described in Phase 5, then feed these structures back into Ontol. This makes Ontol the "backbone" for an intelligent system. Once our bridge is finished, we'll have more insight on how to address the problems from Section 7.1.

In the remainder of this section, we discuss future work for the components of our bridge. We also discuss The Speed Prior, which we suggest as an alternate metric to Ockham's Razor for evaluating the theories proposed by our system.

8.2.1 The Speed Prior

As mentioned in Assumption III in Chapter 1, a useful rule-of-thumb for evaluating a theory developed by a computer is how well it compresses the data. Or more precisely, it's useful to set a description-length prior and maximize the probability of the model given the input data. We somewhat agree with these claims. An intelligent agent should be able to build a model that concisely characterizes its sensor data, and it should be able to use this model to answer queries about the data. Such queries might consist of making accurate predictions about given situations. The agent should also be able to generate plans to accomplish goals (or obtain reward). However, the time needed (in terms of steps of computation) to answer these queries should also be taken into account. Thus, it is sometimes useful to occasionally trade memory for time. For example, an intelligent being might cache a result that it has deduced if it expects to use the result again.

To make this concrete, suppose our agent's domain is Euclidean Geometry. In this domain, a huge but finite set of theorems of can be "compressed" down to a model

containing just 5 postulates and some rules for inference. Such a model would neither be very useful nor would it work the same way as a person. A professional (human) geometer would likely "cache" useful lemmas, thereby speeding up his or her future deductions. It seems true that the same should apply to a generally intelligent being. Another example involves sensor data. If we equip our agent with a video camera, it's possible that the most concise representation of the data (if the pictures are fairly continuous) will be an encoding typical of many video compression algorithms. That is, the representation might fully describe the initial frame, then describe each subsequent frame as changes from its previous frame. A problem with this approach is that it would take longer to answer queries about the end of the day than the beginning (because the entire day would have to be "unwrapped"). This also seems contrary to our intuitions about what an intelligent agent should be able to do.

Thus, we propose using an alternative to Ockham's Razor called "The Speed Prior", first introduced by (Schmidhuber 2002), which states "The quickest model (that predicts the data) is the best model.". By quickest, we mean the model that takes the fewest steps of computation to get accurate answers to queries. Of course, there's a trade-off between speed and accuracy, but this can be folded into a single number by setting a parameter that would act as an "exchange rate" between steps of computation and bits of accuracy. The Speed Prior somewhat overlaps with Ockham's Razor in that fast models tend to be small and tidy so that computation isn't spent searching through disorganized sets of information. The Speed Prior also addresses the utility of caching: caching the answers to frequent queries (or frequent "way points" in derivations) can yield a faster model.

The chief drawback of The Speed Prior is that it's not as straightforward to

evaluate as a measure based on description-length or Equation 3.1. To start, it's not trivial to compute the probability of the data given a model and a fixed amount of computation time. We could take a Monte Carlo approach, where we run the model many times for n steps, and count the number of runs that produced the data, but this might be expensive. Even if we could quickly compute the probability of data given a model and certain amount of computation steps, we'd still have the question of the "exchange rate" between time and accuracy. For example, it's not straightforward whether a model that generates the data with 99% probability using a million steps of computation would be preferable to a model that uses only 1,000 computation steps but gives the data with only 10% probability.

8.2.2 Future work for Ontol

In this dissertation, we've described how Ontol builds and uses an ontology. Our future goal is for Ontol to serve as the backbone for a full cognitive architecture. If, as we argue in Section 6.3, we can represent relational structures as feature-sets, then we can feed these feature-sets into Ontol. So, our plan is to provide Ontol with a strong search and ontology builder, then let Ontol do constraint-satisfaction on these relational structures. We hypothesize that this will be sufficient to do many cognitive tasks such as planning and reasoning. To do this, we must complete the future work discussed below.

Incremental Learning The methods currently described are all batch algorithms. But a lifelong learner will be continually receiving new information, so we'd like our algorithms to run in incremental mode. Following the assimilation/accommodation

model of Piaget in (Piaget 1952) and (Piaget 1954), we suggest an incremental approach where we *parse/explain* (assimilate) each new sensory item as it comes in. After some amount of time, we attempt to formulate new concepts for (or accommodate) unexplained data.

Combined Chunking and Merging In Subsection 3.5.2, we showed how to find candidate equivalence classes, but we don't demonstrate how Chunking and Merging work together to form structures to represent invariant concepts such as that in Figure 2.8. That is, once we have candidate equivalence classes, the question then becomes how to use them to compress the data. To do this, we replace every instance of a feature by the equivalence classes that contain it, and see if this new representation allows for further chunking. Figure 8.1 shows the Zoo Ontology from Figure 3.7, except that this ontology has been chunked, then merged, then chunked again. This structure is too complicated for us to easily evaluate based on its form alone. Conceivably, we can evaluate structures like this by using them to do semi-supervised learning, using the algorithm in Table 3.5, but this is the subject of future work.

A preliminary algorithm for combining Chunking and Merging is shown in Table 8.1. The basic idea is that we first chunk the ontology using The Cruncher, then create context vectors and crunch these to get candidate ORs as described in Subsection 3.5.2. Using the first n ORs generated, we replace each feature with its equivance class. For example, if we find the equivance class `<noun>` as `dog` or `cat` or `orangutan`, then we'll replace each instance of `dog` etc. with `<noun>`. This allows us to find further chunks, so we run The Cruncher again on this "replaced" ontology. The equivance classes allow us to find chunks where we couldn't before.

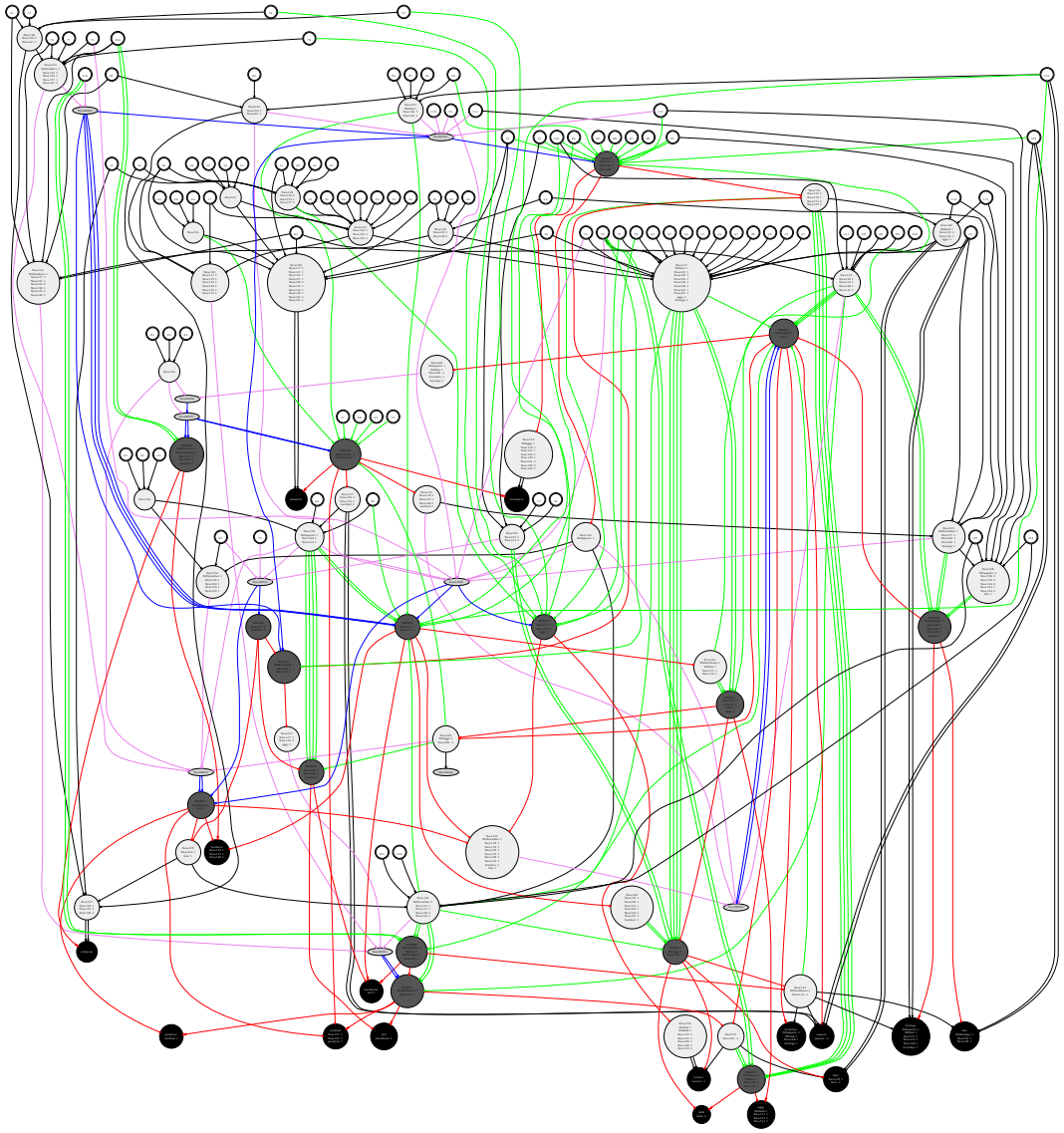


FIG. 8.1. **The Chunked and Merged Zoo Ontology.** Black nodes are raw features, dark grey nodes are ORs and light grey nodes are ANDs.

Table 8.1. **Algorithm for Combined Chunking and Merging**

```
// Returns a new ontology that is both chunked and merged combinedchunkingmerging(S)
  // Keep crunching the ontology, equivalence classes allow us to chunk where we couldn't before
  while E(S) is decreasing
    // Crunch the ontology
    let S = Cruncher(S)
    // Find equivalence classes
    let Q = Merger (S)
    // Replace each reference of a node i with its corresponding equivalence class j
    foreach node i in S
      foreach e in Q
        if i ∈ e then replace i with the name of e
  return S
```

Preliminary results for this approach are shown in Table 8.2. We show the combined Chunking and Merging algorithm for the artificial grammar in Table 3.4, the UCI Zoo dataset, and the UCI mushroom dataset. In each case, we show the ontology's energy $E(\Omega)$ before crunching, after crunching, then after 1 iteration of merging and crunching again. In each case, merging allows us to find chunks where we couldn't before. We also show the top 3 equivalence classes found for each dataset. A full investigation of these ideas is the subject of future work.

Splitting Both Chunking and Merging are "lumping" algorithms. That is, they take existing concepts and put them into groups. Chunking does this using intensions (grouping by the *properties* of concepts), and Merging does this by extensions (grouping by explicitly listing the *instances* of concepts).

It seems clear that people sometimes form categories by finding similarities among a group of concepts, but it also seems likely that people *split* existing concepts to form new concepts. For example, a child might initially lump both cows

Table 8.2. **Preliminary Results from Combined Chunking and Merging.** We show the energy function E for various datasets with crunching alone, and with combined chunking and merging. In each case, merging allows us to further reduce our energy score beyond what we could do through crunching alone. We also show the first 3 discovered equivalence classes for each dataset.

Dataset	Before	Cruncher	Chunking & Merging
mushroom	44,102	11,569	11,447
Artificial Grammar	21,996	13,340	12,200
zoo	13,434	2,473	2,447

Dataset	Top 3 Eq. Classes
mushroom	{CLUSTERED, PENDANT} {FLAT, NO} {BROWN, BULBOUS}
Artificial Grammar	{2A, 2B, 2C, 2D} {1A, 1B, 1C, 1D, 1E, 1F} {3B, 3D}
zoo	{predator, toothed} {breathes, haslegs} {aquatic, fins}

and horses into a single category (which we'll refer to as "cow/horse"s), but later split these into horses and cattle. This type of concept formation, "Splitting", is the subject of future work.

Essentially, Splitting means increasing the resolution of our model for areas where a relatively small increase in resolution yields a large increase in predictive power. For example, from a far distance, a cow and a horse look similar. But its easy to tell a horse from a cow by looking at the bulkiness of the animal's body or the length of the animal's head (or looking for horns or udders). With this added bit of knowledge, we have information about the likelihood that the animal is a fast runner, whether it's involved in meat or dairy production, whether it's ridable, and various pieces of information about the animal's behavior. In the case of cows and horses, it usually pays to invest a small amount of extra resolution in exchange for all the added information from making the split. To address Splitting, we need to address *low-resolution* in Ontol.

Wide Signatures and Low Resolution In Figure 2.8, we show how we might represent the "dog" concept with invariance to translation and rotation. The dog shape on the right of this figure is an AND, which can be thought of as a bag of features such that if all these features are ON, then "dog" is also ON. However, given that the right-side dog is ON, we don't know which raw-sensors are ON. That is, there are many configurations of low-level sensors that might turn the right-side dog ON. So the bag of features is essential a hash or *signature* for the dog concept. We define the "signature width" to be the number of items in the bag of features (in this case, we have 4 items shown).

In general, if a concept has a wider signature, then fewer low-level configurations turn the concept ON. Using this, we can control the *resolution* for a concept by widening or narrowing the signature (i.e., adding or removing items from the bag of features). For example, we can describe a "horse" by making its bag of feature equal to that of a "cow/horse" plus the features of "a slight build" and "a long muzzle". So we can refine concepts by widening their signatures. (Conversely, we can relax or "broaden" a concept by decreasing the number of nodes in its signature.)

So far, we've discussed a mechanism for *how* to refine concepts, but the question remains of *when* to refine or "split" concepts. We suspect that The Speed Bias might be necessary to answer this question. That is, we might have to take computation time (during parsing) into account. We suggest that refinement is done when the predictive power gained by the split outweighs the extra computation needed to determine the values for the extra features in the signature.

Constraint Satisfaction Search The idea of signatures opens up for exploration the area of constraint satisfaction search. Suppose we know that the "dog" concept is ON in Figure 2.8. As we mentioned earlier, there are many possible instantiations of this concept in our low-level sensor nodes.

A problem we might want to look at is finding a configuration that is likely to turn the right-side dog ON. We might also be interested in the *most likely* configuration given that "dog" is ON. We can also further constrain the configurations by adding more concepts that need to be satisfied. For example, if we have another concept elsewhere in our ontology that corresponds to "large thing", and another that corresponds to "single item", we can effectively search for a single large dog by

searching for configurations that turn all 3 of these concepts ON. This way, we can get our computer to "imagine" —mentally create new instantiations of— concepts like a large spotted dog. If our constraints are few, going all the way down and specifying the values of raw sensors might not be fruitful, because these are going to be wildly variable. Further up the ontology won't be as variable though, and we should get to know some properties of "dog". For example, we might learn that pixel1827 is ON half the time. (Which doesn't tell us much.) But there should also be more invariant concepts such as that dogs and cats are rarely seen together.

So, we suggest it would be useful to have an algorithm that takes in the values for a set of "constraint" nodes (in this case that the right-side dog node is ON) and a set of "input" nodes for which we want to know that values (which don't necessarily have to be raw sensor nodes) and returns a configuration of the input nodes that is likely to turn ON the constraint nodes. This search can be non-trivial. Using similar arguments as we used in Subsection 3.4.1, we can express 3-SAT problems in this formalism, so this is an NP-complete problem, but there might be an efficient search that works well for non-pathological cases.

If we have this search, not only can we get our computer to draw dogs, but we argue that we can also do planning. To do this, we need to express the planning problem as a feature-set constraint-satisfaction problem, then use this search, and "translate" the solution back into the planning domain. In future work, we plan to do this by using the MacGlashan Transform.

It might be interesting for this search algorithm to learn macros like a person might. For example, when a person learns to draw, he or she learns some shading and foreshortening tricks that are useful for many drawing tasks. To give another

example, when a baby learns to speak, he or she learns basic phonemes, and then uses these phonemes as macros to build the words and phrases that he or she hears.

Finally, we would like to unify the constraint-satisfaction search algorithm with the parsing/truth-derivation algorithm, since both are basically inference searches.

8.2.3 Future work for Phase 2

There's still much work to be done for Phase 2. For example, we need to address the case where our "retina", instead of being a proper grid, is a random triangulation. In this case, there will be no direct isomorphism from one retinal region to another. Investigation is needed as to whether or not there will be isomorphic behavior further up the respective hierarchies of 2 retinal regions in this case, or whether additional mechanisms will be needed. It seems that at low-resolution, different retinal areas will behave similarly. If this is the case, then we should get both scale and rotational invariance using the methods described in Chapter 4.

Also, as we mentioned in Section 4.4, it might be interesting to investigate whether we can work directly in behavioral signature space for characterizing properties of cortical regions. This space is in feature-set format, the format Ontol works with, so conceivably, we can use Ontol to find patterns in the behaviors of different cortical regions.

We also need to address how our cortical regions are segmented to begin with. Ideally, we'd have a single algorithm that does both the segmentation and finds the behaviorally isomorphic cortical regions. That is, this should actually be a simultaneous search: finding the segments and finding behavioral overlap. Perhaps we can

find a small behavioral overlap between 2 areas, then we "push out" both areas to see how far we can extend the analogy.

Also, if we can then represent relational, parameterized data in terms of feature-sets, we can use Ontol to build a hierarchy of relations, and use these relations to characterize and make predictions. It might be interesting to create a number of cortical regions (either by Ontol or by hand), generate the behavioral signatures, then feed these signatures into another instantiation of Ontol to create an ontology of behaviors.

All these areas are the subject of future work.

8.2.4 Future work for Phase 3

As with all our parts of the bridge, we need to investigate how to smoothly connect Phase 3 to the part before it (Phase 2). It would also be worth investigating the process of finding "primitive" mappings and minimal mapping sets. We also propose a future application that uses Phase 3 for speaker classification and identification by voice. We discuss these ideas further below.

Finding Primitive Mappings and Minimal Mapping Set Given a set of mappings, it might be useful to find a set of "primitive" mappings out of which the rest of the mappings can be composed. For example, we might have one mapping M that rotates a shape 30 degrees, then translates the shape right by 15 pixels, another that rotates 45 degrees and translates 10 pixels to the left, and a large number of other mappings that likewise both translate and rotate a shape. In this case, we can

break this large number of mappings down to a set of 5 primitive mappings such as {Rotate5, TranslateRight5, TranslateLeft5, TranslateUp5, TranslateDown5}. Then, our mapping M from above can be expressed as

$$\begin{aligned} M(x) \quad = \quad & \text{TranslateRight5}(\text{TranslateRight5}(\text{TranslateRight5}(\\ & \text{TranslateRight5}(\text{TranslateRight5}(\text{Rotate5}(\text{Rotate5}(\\ & \text{Rotate5}(\text{Rotate5}(\text{Rotate5}(\text{Rotate5}(x)))))))))) \end{aligned}$$

This is cumbersome because we have so many repeats (this expression of M requires 11 transformations). If our goal is compression, we might want to add to our primitives to make a "minimal" set, where the size of this set plus the size of our descriptions using this set is minimized. Such a set might be

{ Rotate5, Rotate20, Rotate45,
TranslateRight5, TranslateRight20, TranslateRight50,
TranslateLeft5, TranslateLeft20, TranslateLeft50,
TranslateUp5, TranslateUp20, TranslateUp50,
TranslateDown5, TranslateDown20, TranslateDown50 }

Then, mapping M can be expressed using only 5 mappings:

$$M(x) \quad = \quad \text{TranslateRight5}(\text{TranslateRight20}(\text{Rotate5}(\text{Rotate5}(\text{Rotate20}(x)))))$$

The "minimal" set also might allow for quicker computation. It would take 2.2 times as long to compute the result of the "primitive" expression for M compared to the "minimal" expression for M because the former has 2.2 times as many mappings as the latter.

Of course, we'd like for our algorithm to automatically find both the primitive and minimal set of mappings. One approach would be the following: since the mappings are represented as sets, we can treat the mappings as feature-sets, then use Ontol to chunk these and split them into their constituent parts.

Future Application: Using Mappings for Speaker Classification and Identification A future application of these mappings is to do speaker classification and identification from spoken utterances. In our day-to-day life, we recognize familiar people by their voices. We suggest that if we treat the mappings created by Phase 3 as feature-sets, then we can get a computer to do this. For example, if we have a hypothetical person Doug, we might create Doug's voice by taking the "standard" American English voice and apply mappings such as "deep" (which lowers the pitch of the voice), "Male", "gravelly", "southern accent", and many others. These are all mappings. If we treat these mappings as *features* of the voice, then we can use Ontol to create an ontology of voices. We suspect that we can use this ontology to do speaker recognition. Note that, since there's nothing in our system specific to sounds, we should be able to use the same methods for other applications, such as recognizing people by their handwriting.

8.2.5 Future work for Phases 4 and 5

Since Phases 4 and 5 are almost entirely conceptual at the moment, the clear next step is to actually implement and test the ideas for these phases. For example, we should implement a version of The MacGlashan Transform using graphlet kernels. Once this is done, we can experiment with metacognition by representing *cognitive* actions as feature-sets and then feeding these feature-sets back into Ontol.

8.3 Concluding Remarks

Ignore the naysayers; go for it!

–Nils J. Nilsson (Nilsson 1995)

We've provided a framework for learning rich relational theories from raw sensor data, we've explained how we can get a "gisty" theory of integers and other world phenomena, and we've provided the starting point and a research plan for approaching The AI Problem. There are still many holes to fill in, and the most important place to do research is where those holes are. Let's go for it!

REFERENCES

[1] Armstrong, T., and Oates, T. 2007. RIPTIDE: Segmenting data using multiple resolutions. In *Proceedings of the 6th IEEE International Conference on Development and Learning.*

[2] Babbage, C. 1864. *Passages from the Life of a Philosopher.* London: Longman, Green, Longman, Roberts, and Green.

[3] Batchelder, E. O. 2002. Bootstrapping the lexicon: A computational model of infant speech segmentation. *Cognition* 83:167–206.

[4] Bernstein, D. 1999. Reusing Old Policies to Accelerate Learning on New MDPs. *Technical Report UM-CS-1999-026, Dept. of CS, U. of Mass., Amherst.*

[5] Berry, N.; Ko, T.; Moy, T.; Pickett, M.; Smrcka, J.; Turnley, J.; and Wu, B. 2003. Computational Social Dynamic Modeling of Group Recruitment. *Sandia Report SAND2003-8754, Sandia National Laboratories.*

[6] Blake, C., and Merz, C. 1998. UCI Repository of Machine Learning Databases.

[7] Blanz, V. 2006. Face recognition based on a 3d morphable model. In *FG*, 617–624. IEEE Computer Society.

[8] Chomsky, N. 2005. *Rules and Representations (Columbia Classics in Philosophy).* Columbia University Press.

[9] Cohen, P.; Oates, T.; Beal, C.; and Adams, N. 2002. Contentful Mental States for Robot Baby. In *Proceedings of the 18th National Conference on Artificial Intelligence.*

[10] Cohen, P. 1998. Growing Ontologies. *Technical Report 98-20, The University of Massachusetts, Amherst*.

[11] Cole, R. 2001. Automated layout of concept lattices using layered diagrams and additive diagrams. In *ACSC '01: Proceedings of the 24th Australasian conference on Computer science*, 47–53. IEEE Computer Society.

[12] Cooper, G. F. 1990. The Computational Complexity of Probabilistic Inference using Bayesian Belief Networks. *Artificial Intelligence* 42(2-3):393–405.

[13] Dietrich, E. 2000. Analogy and Conceptual Change, or You Can't Step into the Same Mind Twice. *Cognitive Dynamics: Conceptual Change in Humans and Machines* 265–294.

[14] Drescher, G. L. 1991. *Made-Up Minds: A Constructivist Approach to Artificial Intelligence*. Cambridge, MA, USA: MIT Press.

[15] Edelman, S.; Solan, Z.; Horn, D.; and Ruppin, E. 2004. Bridging computational, formal and psycholinguistic approaches to language. In *In Proc. of The 26th Conference of The Cognitive Science Society*.

[16] Ehrig, H.; Engels, G.; Kreowski, H.; and Rozenberg, G., eds. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific Publishing Co.

[17] Elkan, C., and Noto, K. 2008. Learning classifiers from only positive and unlabeled data. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2008)*, 213–220.

[18] Falkenhainer, B.; Forbus, K.; and Gentner, D. 1989. The Structure Mapping Engine: Algorithm and Examples. *Artificial Intelligence*.

[19] Fasulo, D. 1999. An Analysis of Recent Work on Clustering Algorithms.

[20] Fei-Fei, L.; Fergus, R.; and Perona, P. 2006. One-Shot Learning of Object Categories. *IEEE Trans. Pattern Anal. Mach. Intell.* 28(4):594–611.

[21] Fodor, J. 1975. *The Language of Thought*. Harvard University Press.

[22] Forbus, K. 2001. Exploring Analogy in the Large. *The Analogical Mind: Perspectives from Cognitive Science* 23–58.

[23] Friedman, N., and Koller, D. 2003. Being Bayesian About Network Structure. A Bayesian Approach to Structure Discovery in Bayesian Networks. *Machine Learning* 50(1-2):95–125.

[24] Gal, A.; Modica, G.; and Jamil, H. 2004. OntoBuilder: Fully Automatic Extraction and Consolidation of Ontologies from Web Sources. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, 853. Washington, DC, USA: IEEE Computer Society.

[25] Ganter, B., and Wille, R. 1999. *Formal Concept Analysis: Mathematical Foundations*. New York: Springer-Verlag.

[26] Garson, G. D. 1991. Interpreting Neural-Network Connection Weights. *AI Expert* 6:46–51.

[27] Gentner, D. 1983. Structure-Mapping: A Theoretical Framework for Analogy. *Cognitive Science* 7:155–170.

[28] George, D., and Hawkins, J. 2009. Towards a Mathematical Theory of Cortical Micro-circuits. *PLoS Comput Biol* 5(10):e1000532+.

[29] George, D. 2008. How the Brain Might Work: A Hierarchical and Temporal Model for Learning and Recognition. *PhD Thesis, Stanford University.*

[30] Getoor, L.; Friedman, N.; Koller, D.; and Pfeffer, A. 2001a. Learning probabilistic relational models. In Dzeroski, S., and Lavrac, N., eds., *Relational Data Mining.* Springer-Verlag.

[31] Getoor, L.; Friedman, N.; Koller, D.; and Taskar, B. 2001b. Learning probabilistic models of relational structure. In *Proceedings of International Conference on Machine Learning (ICML).*

[32] Getoor, L.; Friedman, N.; Koller, D.; and Taskar, B. 2002. Learning probabilistic models of link structure. *Journal of Machine Learning Research* 3:679– –707.

[33] Gobet, F.; Lane, P.; Croker, S.; Cheng, P.; Jones, G.; Oliver, I.; and Pine, J. 2001. Chunking Mechanisms in Human Learning. *Trends in Cognitive Sciences* 5(6):236–243.

[34] Gold, E. M. 1967. Language identification in the limit. *Information and Control* 10(5):447–474.

[35] Gonzalez, J. A.; Holder, L. B.; and Cook, D. J. 2002. Graph Based Relational Concept Learning. In *Proceedings of the International Conference on Machine Learning.*

[36] Good, I. J. 1965. Speculations Concerning the First Ultraintelligent Machine. *Advances in Computers* 6.

[37] Gruber, T. R. 1993. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In Guarino, N., and Poli, R., eds., *Formal Ontology in*

Conceptual Analysis and Knowledge Representation. Deventer, The Netherlands: Kluwer Academic Publishers.

[38] Harnad, S. 1990. The Symbol Grounding Problem. *Physica D* 42:335–346.

[39] Hawkins, J., and Blakeslee, S. 2004. *On Intelligence*. Times Books.

[40] Hofstadter, D. R. 1984. The Copycat Project: An Experiment in Nondeterminism and Creative Analogies. *Massachusetts Institute of Technology* 755.

[41] Hofstadter, D. R. 2001. Analogy as the Core of Cognition. *The Analogical Mind: Perspectives from Cognitive Science* 499–538.

[42] Holder, L.; Cook, D.; and Djoko, S. 1994. Substructure Discovery in the SUBDUE System. In *Proceedings of the Workshop on Knowledge Discovery in Databases*.

[43] Hölldobler, B., and Wilson, E. O. 1998. *Journey to the Ants: A Story of Scientific Exploration*. Belknap Press.

[44] Hopcroft, J., and Wong, J. 1974. A Linear Time Algorithm for Isomorphism of Planar Graphs. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, 310–324.

[45] Hoppner, F.; Klawonn, F.; Kruse, R.; and Runkler, T. 1999. *Fuzzy Cluster Analysis*. Wiley.

[46] hua Zhou, Z.; chuan Zhan, D.; and Yang, Q. 2007. Semi-supervised learning with very few labeled training examples. In *Twenty-Second AAAI Conference on Artificial Intelligence (AAAI-07*, 675–680.

[47] Hummel, J. E., and Holyoak, K. J. 2005. Relational Reasoning in a Neurally Plausible Cognitive Architecture: An Overview of the LISA Project. *Current Directions in Psychological Science* 14(3):153–157.

[48] Hummel, J. E., and Holyoak, K. J. 2006. A Symbolic-Connectionist Theory of Relational Inference and Generalization. *Psychological Review*.

[49] Hutter, M. 2004. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*. EATCS. Berlin: Springer.

[50] Iba, W., and Langley, P. 2001. Unsupervised Learning of Probabilistic Concept Hierarchies. *Lecture Notes in Computer Science* 2049:39.

[51] Kanerva, P. 1988. *Sparse Distributed Memory*. Cambridge, MA, USA: MIT Press.

[52] Karp, R. M. 1972. Reducibility Among Combinatorial Problems. In Miller, R. E., and Thatcher, J. W., eds., *Complexity of Computer Computations*. Plenum Press. 85–103.

[53] Kemp, C., and Tenenbaum, J. B. 2008. The discovery of structural form. *Proceedings of the National Academy of Sciences of the United States of America*.

[54] Kuipers, B. 2008. Drinking from the Firehose of Experience. *Artificial Intelligence in Medicine* 44(2):155–170.

[55] Lenat, D., and Guha, R. V. 1990. *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project*. Addison-Wesley.

[56] Leslie, A. M., and Keeble, S. 1987. Do Six-month-old Infants Perceive Causality? *Cognition* 265–288.

[57] Little, J. D. C. 1955. The Use of Storage Water in Hydroelectric Systems. *Opns Res.* (3):187–197.

[58] Marcus, G. F. 1993. Negative Evidence in Language Acquisition. *Cognition.*

[59] Marshall, J. B. D., and Hofstadter, D. R. 1996. Beyond Copycat: Incorporating Self-Watching into a Computer Model of High-Level Perception and Analogy-Making. In *Online Proceedings of the 1996 Midwest Artificial Intelligence and Cognitive Science Conference.*

[60] McKay, B. 1981. Practical Graph Isomorphism. *Congressus Numerantium* 30:45–87.

[61] Meraz, R. F.; He, X.; Ding, C. H. Q.; and Holbrook, S. R. 2004. Positive sample only learning (psol) for predicting rna genes in e. coli. In *Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference*, CSB '04, 535–538. Washington, DC, USA: IEEE Computer Society.

[62] Muggleton, S. 1994. Inductive Logic Programming: Theory and methods. *The Journal of Logic Programming* 19-20:629–679.

[63] Muggleton, S. 1996. Learning from positive data. In Muggleton, S., ed., *Inductive Logic Programming Workshop*, volume 1314 of *Lecture Notes in Computer Science*, 358–376. Springer.

[64] Nagel, K., and Schreckenberg, M. 1995. Traffic Jam Dynamics in Stochastic Cellular Automata. *Los Alamos National Laboratories technical report.*

[65] Nevill-Manning, C. G., and Witten, I. H. 1997. Compression and Explanation Using Hierarchical Grammars. *The Computer Journal* 40(2/3):103.

[66] Newell, A., and Simon, H. A. 1972. *Human Problem Solving.* Prentice Hall.

[67] Nilsson, N. J. 1995. Eye on the Prize. *AI Magazine* 16(2):9–17.

[68] Oates, T.; Doshi, S.; and Huang, F. 2003. Estimating Maximum Likelihood Parameters for Stochastic Context-Free Graph Grammars. In *The 13th Annual International Conference on Inductive Logic Programming.*

[69] O'Donoghue, D. 2005. Finding Novel Analogies. *PhD Thesis, University College Dublin.*

[70] Olson, C. F. 1995. Parallel algorithms for hierarchical clustering. *Parallel Computing* 21(8):1313–1325.

[71] Pearl, J. 1985. Bayesian networks: A model of self-activated memory for evidential reasoning. In *Proceedings of the 7th Conference of the Cognitive Science Society, University of California, Irvine*, 329–334.

[72] Pearl, J. 2000. *Causality.* Cambridge University Press.

[73] Perruchet, P., and Vintner, A. 1998. PARSER: A Model for Word Segmentation. *Journal of Memory and Language* 39:246–263.

[74] Piaget, J. 1952. *The Origins of Intelligence in Children.* International Universities Press.

[75] Piaget, J. 1954. *The Construction of Reality in the Child.* Basic Books.

[76] Pickett, M., and Barto, A. 2002. PolicyBlocks: An Algorithm For Creating Useful Macro-Actions in Reinforcement Learning. In *Proceedings of the International Conference on Machine Learning.*

[77] Pickett, M., and Oates, T. 2005. The Cruncher: Automatic Concept Formation Using Minimum Description Length. In *Proceedings of the 6th International Symposium on Abstraction, Reformulation and Approximation (SARA 2005)*, Lecture Notes in Artificial Intelligence. Springer Verlag.

[78] Pickett, M.; Miner, D.; and Oates, T. 2007. A Gauntlet for Evaluating Cognitive Architectures. In *Working Notes of the AAAI Workshop on Evaluating Architectures for Intelligence*.

[79] Pickett, M.; Miner, D.; and Oates, T. 2008. Essential Phenomena of General Intelligence. In *Proceedings of The First Conference on Artificial General Intelligence*.

[80] Pierce, D., and Kuipers, B. 1997. Map learning with Uninterpreted Sensors and Effectors. *Artificial Intelligence* 92:169–229.

[81] Pinker, S. 1994. *The Language Instinct*. Harper Perennial Modern Classics.

[82] Plato. 360 BC. *The Republic, Book VII*. (Public domain).

[83] Plato. 387 BC. *The Meno*. (Public domain).

[84] Przulj, N. 2007. Biological Network Comparison Using Graphlet Degree Distribution. *Bioinformatics* 23(2):e177–e183.

[85] Raina, R.; Battle, A.; Lee, H.; Packer, B.; and Ng, A. Y. 2007. Self-taught learning: Transfer learning from unlabeled data. In *Proceedings of the 24th International Conference on Machine learning*, 759–766.

[86] Riesenhuber, M., and Poggio, T. 1999. Hierarchical Models of Object Recognition in Cortex. *Nature neuroscience* 2(11):1019–1025.

[87] Robinet, V., and Lemaire, B. 2009. MDLChunker: a MDL-based Model of Word Segmentation. In *Proceedings of the 31th Annual Conference of the Cognitive Science Society*.

[88] Rohde, D. L. T., and Plaut, D. C. 1999. Language Acquisition in the Absence of Explicit Negative Evidence: How Important is Starting Small? *Cognition*.

[89] Rosenbloom, P.; Laird, J.; and Newell, A. 1993. *The SOAR papers: research on integrated intelligence*. Cambridge, MA: MIT Press.

[90] Rosenstein, M., and Cohen, P. 1999. Continuous Categories for a Mobile Robot. In *proceedings of the 16th National Conference on Artificial Intelligence*.

[91] Schmidhuber, J. 2002. The speed prior: A new simplicity measure yielding near-optimal computable predictions. In Kivinen, J., and Sloan, R. H., eds., *COLT*, volume 2375 of *Lecture Notes in Computer Science*, 216–228. Springer.

[92] Servan-schreiber, E., and Anderson, J. R. 1990. Learning artificial grammars with competitive chunking. *Journal of Experimental Psychology: Learning, Memory, and Cognition* 16:592–608.

[93] Shannon, C. E. 1948. A Mathematical Theory of Communication. *Bell System Technical Journal* 27:379–423 and 623–656.

[94] Shervashidze, N.; Vishwanathan, S.; Petri, T.; Mehlhorn, K.; and Borgwardt, K. 2009. Efficient Graphlet Kernels for Large Graph Comparison. In *12th International Conference on Artificial Intelligence and Statistics (AISTATS)*. Clearwater Beach, Fl, USA, April, 16-18, 2009: Society for Artificial Intelligence and Statistics.

[95] Skiena, S. 1990. Graph Isomorphism. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica* 181–187.

[96] Stolcke, A., and Omohundro, S. 1994. Inducing Probabilistic Grammars by Bayesian Model Merging. In Carrasco, R., and Oncina, J., eds., *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 106–118.

[97] Sun, R. 2004. Desiderata for Cognitive Architectures. *Philosophical Psychology* 17(3):341–373.

[98] Sutton, R., and Barto, A. 1998. *Reinforcement Learning: An Introduction*. MIT Press.

[99] Thompson, K., and Langley, P. 1991. Concept formation in structured domains. *Concept Formation: Knowledge and Experience in Unsupervised Learning (D. H. Fisher and M. Pazzani, editors)* Chapter 5.

[100] Thrun, S., and Schwartz, A. 1995. Finding Structure in Reinforcement Learning. In *Advances in Neural Information Processing Systems*.

[101] Velardi, P.; Fabriani, P.; and Missikoff, M. 2001. Using text processing techniques to automatically enrich a domain ontology. In *FOIS '01: Proceedings of the International Conference on Formal Ontology in Information Systems*, 270–284. New York, NY, USA: ACM Press.

[102] von der Malsburg, C. 1999. The What and Why of Binding: The Modeler's Perspective. *Neuron* 24:95–104.

[103] Witbrock, M.; Matuszek, C.; Brusseau, A.; Kahlert, R. C.; Fraser, C.; and Lenat, D. 2005. Knowledge Begets Knowledge: Steps towards Assisted Knowledge Acquisition in Cyc. In *Papers from the 2005 AAAI Spring Symposium on Knowledge Collection from Volunteer Contributors (KCVC)*, 99–105.

[104] Wolff, J. G. 2003. Information Compression by Multiple Alignment, Unification and Search as a Unifying Principle in Computing and Cognition. *Artif. Intell. Rev.* 19(3):193–230.

[105] Zhu, X. 2008. Semi-Supervised Learning Literature Survey. *Technical Report, University of Wisconsin Madison, Computer Sciences TR 1530.*

[106] Ziv, J., and Lempel, A. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23(3):337–343.