Associational Computing in Language Processing and Robotics

Thomas S. Stepleton

May 2, 2002

Abstract

This thesis describes two summers of research into a general technique for knowledge representation and machine learning in which individual concepts are represented as massively interlinked nodes within a weighted, directed graph data structure. Paralleling contemporary research, the first application of this technique was the representation of word meanings. Analysis techniques involving clustering and neural networks were employed to explore the nature of these representations. Later, this technique was adapted to the learning and execution of tasks in real-time by robots. Though it was not able to fully master tasks requiring state, the technique showed enough promise in learning simpler tasks to merit further investigation.

1 Introduction

"Associational computing" in this text refers to a body of ideas with an intuitive appeal and a long history. Essentially, it proposes a set of concepts linked together by semantic associations of some sort; computations performed over such a structure manipulate these associations to derive and store information. For many, this should be highly reminiscent of the mechanism of thought hypothesized by David Hume, which, for the sake of subsequent contrast, is partially reproduced below:

> ...'tis impossible the same simple ideas should fall regularly into complex ones (as they commonly do) without some bond of union among them, without some associating quality, by which one idea introduces another...
>
> ...The qualities, from which this association arises, and by which the mind is after this manner conveyed from one idea to another, are three, *viz.* RESEMBLANCE, CONTIGUITY in time or place, and CAUSE and EFFECT. ([1], 1.iv)

Hume's formulation and subsequent associationist interpretations of thought have fallen out of favor for a number of reasons. Among them, the most important is that no associationist scheme has come up with an effective model for mental processes that seem to rely heavily on the syntactic manipulation of constituents. The production and understanding of language is perhaps the most familiar of these processes, and in this domain one of the most formidable associationist edifices ever constructed, B.F. Skinner's *Verbal Behavior*, was famously demolished by Noam Chomsky for precisely this reason ([2], 55-57). No amount of stimulus-response apparatus—no system of learned links between experienced concepts—could account for the sheer versatility of our language ability. Rightfully, most research gradually abandoned the search for associationist accounts of human cognition (especially vis-a-vis language processing) and adopted in whole or in part the viewpoint of the mind as a symbol processor, manipulating symbol tokens according to task-specific grammars. Soon, in the artificial intelligence world, efforts were undertaken to construct analogous computer programs. These efforts continue today, the largest being perhaps the Cyc project [3], which aims to represent millions of statements of everyday knowledge in a custom propositional logic.

No current research is engaged in trying to implement the old Humean model of cognition in software. Operant conditioning happens to be enjoying a recent resurgence in robot control, mostly because (as noted by Touretzky and Saksida in [4], 2) its success in animal training might suggest a better means of imparting tasks to robots than Q-learning and other training methods. Nevertheless, there are efforts underway in artificial intelligence, robotics, and psychology that employ associational computation in more novel, if unproven, ways. These new projects, to be described in detail later in this text, differ in a few crucial details from traditional associationist schemes. First, the links between concepts, while still reflecting some aspect of semantic relatedness, are seldom as abstract and subjective as the three proposed by Hume. Second, these links are often far greater in number than what one would normally expect, often connecting (if weakly) concepts with seemingly little in common rather than a sparse few with immediate semantic relatedness. Third, computations that make use of these later model semantic networks often do so by analyzing *all* of the links held by a concept and looking for distinctive patterns.

This last point is of critical importance and represents a fundamental departure from Hume's formulation. Instead of treating associations strictly as an interconceptual rail system for trains of thought, where thought processes follow individual links from one concept to another, these new approaches impose no real restrictions on how the data encoded in semantic networks are to be used. Any type of analysis may be used to derive information from concepts, clusters of concepts, or the entire network.

This general characterization of these novel semantic networks is detailed enough to permit the definitions of terms

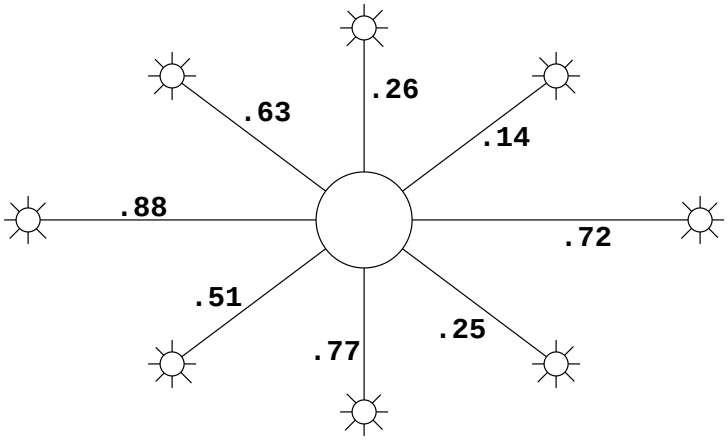

Figure 1: The large node in the center of the image depicts a *represented concept*. It is connected by *relations* bearing varying *relatedness values* to nodes representing *related concepts*.

that will be used to describe them throughout this paper.

- Because the networks are so easily realized by weighted graph data structures, they will often be referred to here as **concept graphs** or simply **graphs**.
- A weighted edge connecting one concept with another is termed a **relation**.
- The weight on a single edge connecting one concept to another is referred to as a **relatedness value**.
- When discussing a particular node in a concept graph, the concept it represents is the **represented concept**.
- Meanwhile, another node connected to the node being presently discussed represents a **related concept**. This term doesn't imply that it bears any strong relation, just that there is an edge emanating to it from the present node.

Figure 1 should help explicate these relationships visually.

With this nomenclature established, there is room to proffer one final philosophical remark. It should be evident that the symbols that are the currency of this system—these massively interlinked nodes—are different from symbols employed in most traditional computational tasks. Typically a symbol is bound to an idea by fiat, an act comprised solely of the assignment of a unique identifier. The meanings of these symbols are later fleshed out by the relationships defined between them, be they logical expressions, state transitions, or other possibilities. In the newer associational

methods explored here, a symbol is bound to an idea by being composed of a large set of relations uniquely reflecting some characteristics of the concept. Naturally the boundary between these two approaches is not well-defined: certainly a symbol in a predicate calculus system could be, broadly speaking, composed of all of the statements containing it. Nevertheless, the extents to which both systems are typically broken down to their fundamental quanta differ. Statements in a predicate calculus are disassembled and analyzed piecemeal, and of all that contain a particular symbol, applications in which only a fraction are of interest or relevance are not uncommon. In contrast, very few applications using the newer associational techniques analyze only portions of the nodes they use to represent ideas. Concepts are compared and computed by operations performed on the entire set of relations emanating from a node. For the most part, then, these nodes, like names within a formal symbol system, are atomic.

These new techniques thus afford a certain degree of philosophical satisfaction. Symbols are still treated as atomic, impermeable objects. Nevertheless, they achieve their binding because they are embedded in the proper place within a web of other symbols, a web that reflects the network of associations and relations between the actual, represented concepts.

2 Word representation

The work of the Summer 2000 research was applying this framework to the representation of word meanings. Several large graphs, each representing some 6,000 words and containing on the orders of 1,000 and 10,000 relations per word, were represented as simple adjacency matrices. The relations employed were rough approximations of word similarity based on frequency of near co-occurrence between represented words and related words in a large body of text. Serendipitously, this rather effective approximation caused much of this work to be nearly identical to a psycholinguistics simulation technique called Hyperspace Analog to Language (HAL) [15]. However, later work exploring the use of neural networks with these word representations did represent apparently novel investigation.

This section describes the creation of the word representation graphs and three kinds of analysis performed upon them: a series of simple comparative procedures between individual nodes, a cluster analysis performed over an entire graph, and the training of neural networks to detect category membership within individual nodes.

2.1 Background

The representation of words has received a great deal of attention from computational linguistics, psycholinguistic modeling, and other fields. The most famous and successful effort to date is likely WordNet [5], a voluminous hand-made database of words structured and cross-referenced according to synonymy, category membership, and numerous other types of relations. WordNet's structure is based on contemporary psycholinguistic hypotheses about how information about words is organized within the brain—the relationships and categories used among items in the database, including synonymy, hyponymy/hypernymy,[1] and part of speech information, reflect similar distinctions humans employ when dealing in word utterances and their meanings. Depending on its part of speech, a single sense of a word within the WordNet database thus consists of links to synonymous word senses, sub- and super-categorical word senses, and other specialized associations, as well as a brief English gloss about the meaning of the sense.

Despite all of this relational information, WordNet does not claim to represent word meanings. Miller et al. point out that no strong psychological hypothesis of word meaning representation exists at present, thus the inclusion of the definitional glosses with word sense information. Certainly a great deal of information about a sense can be deduced from the associations presented in its database entry, perhaps even enough in many cases to uniquely identify its referent—in a practical sense, its meaning. Nevertheless, its true meaning, in the eyes of the WordNet researchers, is deferred to the English speaking user.

Word meaning, then, is left to a burgeoning menagerie of representational schema, and whether the information WordNet provides suffices for this task is not clear. Some schema provide more, others far less. One approach described by Kevin Knight represents individual word senses as small graphs with nodes roughly corresponding to placeholders for linguistic constituents (e.g. noun and verb phrases) and edges roughly corresponding to thematic roles in the word's semantic lexical entry [6]. A library of combination techniques can be used to synthesize these graphs into larger ones representing whole phrases and to deduce the meaning of unknown words from a short, user-supplied description. Meanwhile, Cynthia Thompson has done a great deal of work matching words to their nodes within semantic tree representations of sentences, though these nodes, the real items representing words, contain only a few synonyms or hypernyms [7]. Along with WordNet, these approaches represent some of the more symbol-based approaches to word representation, and there are many others.

Naturally there are many connectionist or non-symbolic approaches to word meaning representation, though these

[1] *Hyponym* and *hypernym* are lexicographic terms that describe category membership relationships. Examples make their meaning clear: *bear* is a hyponym of *mammal* because a bear is a member of the mammal category; meanwhile, *mammal* is a hypernym of *bear* because the mammal category contains all bears.

often tend to entail not much on the surface beyond creating and combining unique symbolic representations for words and phrases. Still, certain crucial aspects of word meaning do tend to work their way into the systems. Jeffrey Elman's 1990 paper on the use of recurrent neural networks in representing time contains a section describing research in which neural networks were trained to recite simple sentences one word after the other (i.e. when given a unique bit sequence representing a single word, the neural network was trained to predict the next word in the sentence). The bit strings themselves reflected no aspect of the words' meanings, though the neural network can be said to have constructed some internal encodings of the sentences during training. Indeed, hidden layer activations recorded during the recitation task share similarities based on the part of speech of the word being processed, thus suggesting that the network has learned some aspects of the words' meanings [8]. Similar meaning-related clustering occurred in the RAAM research of Blank, Meeden, and Marshall, in which a neural network was used to successively compress a sequence of words in a sentence into a single distributed representation. Not only did these representations tend to cluster according to similarities in sentence meanings, but also the average activations of the hidden layer clustered according to the input words' part of speech and category membership [9]. Mikkulainen and Dyer generated word representations by training a neural network to take a collection of activation patterns representing words in a sentence and place the same patterns in output slots assigned to certain semantic roles. Interestingly, the activation patterns representing words input into the network were allowed to be changed by backpropagation as the network evolved. When network and word representations converged on a final configuration, the words tended to cluster together based on certain categories they belonged to [10].

One other non-symbolic representation scheme of note is Tony Plate's Holographic Reduced Representations (HRRs) [11]. Similar to the matrix and tensor representation work of Smolensky [12] and others, and like the RAAM research of Blank et al., HRRs are able to compress complex hierarchical data structures into a single high-dimensional vector of real numbers. The compression technique isn't a neural network, however, but a combination technique called circular convolution. Extraction of embedded information is done with a similar technique entitled circular correlation. Plate demonstrated the possibility of encoding sentences within HRRs and showed that structurally similar sentences had similar representations (measured by the size of the dot-product of the representations). However, since vectors representing terminal nodes (i.e. words) in these data structures were arbitrary vectors and since the combination technique employed is general to all data, no real representation of the semantics of words or sentences exists in this scheme. This contrasts with RAAM, where the neural network appears to have optimized itself and its representations to the particular demands of word and sentence representation, taking advantage of semantic categories

as they presented themselves.

Finally, lying somewhere in the midst of the symbolic and non-symbolic word representation schema outlined above are those approaches that represent words as large collections of statistical data on their usage. Building on the work of Lofti Zadeh, Burqhard Rieger is one of the early adopters and advocates of this approach. In a well-cited 1980 paper, he describes a complicated statistical mapping between words in German newspapers and accumulated "usage regularities," which can be taken to mean the various instances of co-occurrence between words [13]. These mappings are expressed as points in a many-dimensional space, and thus, using a provisional distance measure, Rieger compares words based on their representational similarity and achieves groupings roughly in accordance with the appraisal of native speakers of German. In later papers, Rieger uses cluster analyses to group similarly constructed word representations into "connotative clouds," a fuzzy analog to sharply-delineated semantic categories [14]. Rieger also proposed in his 1980 paper that his representation strategy, being that it was a practical refinement of a framework developed by Zadeh, permitted word concepts to be combined and manipulated with fuzzy set operations like union, intersection, and difference.

In the mid to late 1990s, two similar representational techniques were developed by psycholinguistics researchers interested in creating models for word concept representation in memory. Like Rieger's work, both approaches represent words as many-dimensional vectors compiled from data on word co-occurrence.[2] The simplest of these techniques is known as Hyperspace Analog to Language, or HAL. The word vectors in HAL are composed of simple co-occurrence scores describing how frequently a represented word appears in close proximity to all the other words in a text corpus. These scores are created by moving a "window" one word at a time over a large body of text. The word in the center of the window is the object of investigation, and in its representational vector the co-occurrence scores of all the other words in the window are incremented. This incrementation is proportional to the distance of the neighbor word from the window's center—more distant words are deemed to be less related and are given a smaller boost in score. Once the scanning is completed, each word in the corpus has its own representational HAL vector with elements corresponding to every other word. What's more, each element at a particular index in each vector corresponds to the same related word. To reduce computational complexity, it is possible to isolate a number of principle components of the parameter space and use those to construct fewer-dimensional word representations. Researchers Burgess and Lund, who describe this technique in their 1996 paper [15], also devote a great deal of study to finding out the most effective number of dimensions for word representations. Finally, similarity between words can be estimated

[2]The subsequent few were adapted from a paper I wrote for my Psychology of Language course.

Context	Dickens, Charles *Martin Chuzzlewit*	Stepleton, Thomas *Searle's Chinese Room: A Critique*	U.S. Internal Revenue Service *Form 1040EZ*	Chesterton, G.K. *Orthodoxy*
Number of occurrences	5	37	0	2

Figure 2: This vector is a hypothetical LSA representation for the word *ludicrous*. The contexts in this example are whole texts, and the values represent how frequently *ludicrous* appears within each.

by simply calculating the Euclidean distance between their representations—the lower the distance, the more related the represented words.

LSA is more complex than HAL, and consequently is able to represent a greater range of concepts. As summed up by researchers Landauer, Foltz, and Laham in [16], it requires the input corpus to be divided up into a collection of contexts as opposed to using a moving context window. There is no prescription on the size of these contexts—they can be individual sentences, paragraphs, chapters, or even entire books. Once segmented, vectors, each representing target words, are created. Each position in the vector corresponds to a different context, and the value in a position simply indicates how often that vector's target word appeared in the position's corresponding context. A hypothetical example of such a vector appears in Figure 2.

So far, these vectors seem fairly similar, at least in spirit, to HAL vectors. However, LSA might seem subject to an additional problem. Most words are common enough that HAL scanning will likely find a number of instances of their co-occurrence roughly proportional to their relatedness, especially if if scans multiple texts. With LSA, however, it is possible that a word might be strongly semantically related to a context and yet never appear within it. An example of such a context might be a pirate story that never uses the word *pirate*, only *buccaneer*—in the vector representing *pirate*, this context would score a dissatisfying zero. Fortunately, and distinctly, LSA uses matrix operations to skirt this problem. Several word representation vectors, stacked on top of each other, form the rows of a large "word by content" matrix, which can then be mathematically decomposed into the product of three matrices using a technique called singular value decomposition. This technique does not destroy any information, and the three matrices can be multiplied to achieve the original matrix. However, when they are first artfully pared to a fraction of their size, deleting some of the information, and then multiplied together again, a valuable approximation of the original matrix is formed. Significantly, words that did not appear at all in certain contexts now have a contextual appearance score roughly proportional to the semantic relatedness between the words and the contexts. Using the above example, *pirate* would now score favorably within the *buccaneer* story context. The complex mathematical reasons for this seemingly

magical phenomenon are described in Landauer et al.'s paper.

LSA's greater versatility when compared with HAL stems from its ability to represent contexts in addition to words; these representations are simply the column vectors from the word by content matrix. LSA also uses a different distance metric from HAL, the cosine of two representational vectors. Similar concepts will have a relatedness score near 1; dissimilar ones will have a score near -1.

HAL, LSA, and perhaps to a lesser extent Rieger's research all employ what will henceforth be referred to as the *co-occurrence hypothesis*, the notion that if a word appears frequently within a certain context, there likely exists some semantic relation between the word and that context. Adapted slightly for HAL, the hypothesis posits that two words frequently occurring in close proximity to each other also have some semantic relationship. This assumption has proven to be quite powerful and indeed was the inspiration and the engine behind the research described here.

2.2 Research introduction and history

It is necessary to mention that a fair portion of the word representation research performed during the summer of 2000 replicates the work of Burgess and Lund. Much of the experimentation was done without knowledge of their efforts, and while more rigorous background research should have uncovered this redundancy, experimental psycholinguistics was not an obvious place to search for prior research. In any case, philosophically and methodologically there are sufficient differences in these two approaches to word representation and manipulation for both to be worthwhile in their own right.

A brief historical digression on the way the project came to emulate HAL is appropriate. For several years, the author had wanted to attempt representing a set of concepts with the massively interlinked representations described in the introduction. The challenge of qualitatively judging millions of relatedness values was considered too daunting until a friend proposed the following related procedure for determining which of two categories contained a particular concept (e.g. determining whether a bear [concept] was a mammal [category 1] or a fish [category 2]):

1. Using a leading search engine, ascertain the number of webpages containing the words for the concept and the first category.
2. Repeat for the second category.
3. Chose whichever category pairing garnered the most hits.

This rather transparent application of the co-occurrence hypothesis works surprisingly well despite many patholog-

ical cases.[3] Thus, it was recognized immediately that it represented a means of automatically generating relatedness values between concept nodes representing words. Each value was simply the fraction of all pages containing the represented word that contained the related word as well. As soon as time permitted, a distributed application was written to automate and expedite the 625,000,000 web queries needed to generate representations for a well-sized words database.

The moral turpitude of this endeavor was understood, and after a quarter of the queries had been executed in the course of a month it was no surprise when the search engine put a stop to it. Nevertheless, it demonstrated the practicality of using the co-occurrence hypothesis in generating a massively interlinked database of concepts. Before long, a considerably more formal effort involving the creation and analysis of a second database from locally-stored text files was started, this time as an established research project funded by the Swarthmore College chapter of Sigma Xi.

2.3 Word representation database creation

The search engine debacle aside, representing word meanings remains an attractive first task for conceptual representation graphs given the abundance of text available over the Internet and the relative ease of manipulating ASCII strings. What's more, words have several well-defined, easily distinguishable characteristics (e.g. lexicographic ones) that many other problem domains lack and seem to be well suited to quick qualitative analysis (such as determining whether two words are actually synonyms).

Without relying on search engine results, another manifestation of the co-occurrence hypothesis was required. The decision to use locally stored text files precluded any easy contextual segmentation, since these files were often very large and irregularly formatted. Rather than try and detect segments like paragraphs and sentences automatically, then, it was decided that it would be easier to do away with them altogether. Instead, a sliding window would serve as an ad-hoc context. A pair of words now needed only to be within a certain range of each other to exhibit a nonzero relatedness value. Whether they appeared in different sentences, paragraphs, or even texts was unimportant; what mattered was that the scanning program found the second word in the close vicinity of the first.

The program written to create word representations was designed to scan a single large text file and generate a representation for a single word. When it found an instance of the target word, it scanned for neighbor words up to

[3] Consider using this method to determine whether an apple was a fruit or a computer, or whether a kangaroo was a captain or a marsupial. In cases like these, the wrong answer wins by a large margin, thanks to the frequent conjunction of certain categorically unrelated words. Better results can be achieved by searching for identity statements constructed from the search terms, such as "apple is a fruit" and "apple is a computer."

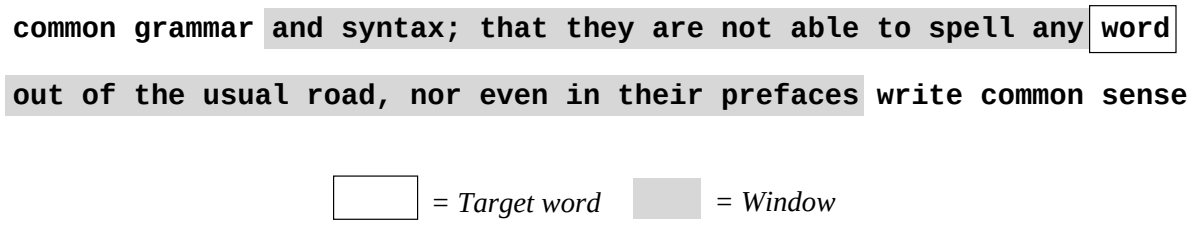

Figure 3: The text scanner at work. After finding an occurrence of the target word *word*, it finds all the other words in the 21-word window and increments corresponding tallies of how frequently they occur in such close contexts.

ten words ahead of the instance and ten words behind. Twenty-one words (or "ten words away") was not an entirely arbitrary choice for the window size; it's large, but not exceedingly large, and hence it was hypothesized that it would uncover a decent variety of related words without including too many unrelated ones. Furthermore, the search engine from the original database experiment featured a 'NEAR' query operator that returned only web pages where two search terms were within ten words of each other. This was in fact what was used by the distributed query application instead of 'AND'.

If the system found within the window any of the words slated for inclusion in the final words database (in this case, the 25,027 words in the Solaris 2.5 /usr/dict/words spellchecking dictionary), it incremented a tally of how frequently the word had co-occurred with the target word within the ten word window. This process is depicted graphically in Figure 3. Once the entire text file was scanned, the program divided all the co-occurrence tallies by the total number of instances of the target word, in effect generating a list of probabilities that their corresponding words would occur within ten words of any instance of the target. These probabilities, thanks to the co-occurrence hypothesis, are estimated relatedness values. All are saved into an ASCII text file.

This method appears nearly identical to the HAL method described in Burgess and Lund's work. The only significant difference is the lack of weighting in incrementing the co-occurrence tallies for words in the scanning window. Burgess and Lund gave a higher score to words that appeared closest in proximity to the target word. While there might be an empirically discoverable distribution of incrementation values that yield the best result, it might not necessarily be one such as theirs, which seems as if it might give even more disproportionally high relatedness scores to words like *the* than the model described here. Overall, the windowing method appears to be fairly robust, gracefully ignoring occasional anomalies in the contexts spanned by the moving window by dint of the far greater volume of well-mannered text it encounters. Indeed, those words that did manage to be created by the earlier search engine experiment appeared in provisional tests to be of considerable quality as well, despite using a much more coarse de-

lineation of contexts. Window size and weighting aren't altogether unimportant—in [15] Burgess and Lund find that a window size of 8 words creates word representation vectors whose Euclidean distances from each other correlate well with human reaction times in word priming experiments. Nevertheless, the co-occurrence hypothesis seems to do pretty well on its own without the benefit of strict window sizing and weighting.

In creating the scanning program, no effort was made to account for the morphology of the target words; plural nouns, common past tense verbs, and common third-person singular verbs were all ignored unless they appeared in the spellchecking dictionary, which limited itself to irregular forms not covered under the simple morphological rules in the Solaris `spell(1)` spellchecking program.[4]

A collection of text corpora for this program was obtained by downloading the (then) entire collection of free and uncopyrighted texts maintained by Project Gutenberg, available at `http://www.gutenberg.net`. At the time of the download (circa July 1999), this totaled to over 2,000 files with a combined size of over a gigabyte. After eliminating a few texts that would be unhelpful to the project (such as *One Divided By pi (to 1 million digits)* by Yasumasa Kanada), the remaining files were concatenated into one very large file. No effort was made to remove the lengthy informational header prepended to all Project Gutenberg texts. In contrast, Burgess and Lund used a large compilation of Usenet posts, a corpus featuring considerably more contemporary language and subject matter.

Because it was suspected that scanning for all the words in the dictionary would take a very long time, another program was written to determine at what frequency dictionary words appeared in the Project Gutenberg corpora. (This program also took no account of word morphology.) It was thought that scanning for all the dictionary words would take a long time, so these results, sorted in descending order, set the schedule for word representation creation. This ensured that research could be carried out on representations of more useful, common words while the scanner simultaneously processed more esoteric ones. The word frequency information created by this program was also used by the scanning program for the final division step described above.

The word scanning program ran for over three months on a 150 megahertz Pentium PC running the Linux operating system. A simple Bourne shell script invoked it sequentially on every word in the frequency-sorted dictionary list. After the creation of 6,336 words, the decision was made to terminate the scans because the frequency of dictionary words had become low enough (around 200 instances in the entire corpora) might have caused aberrations in the co-occurrence information.[5]

[4] The program admits words like **munchize* and **tacoist*.

[5] An example of such an aberration comes from a much more common word, *project*, which bore a very high relatedness value to the word *gutenberg* due to the informational header. For these more infrequent words, it was feared that authors who favored them would show particular usage habits that did not necessarily reflect the meaning of the actual word (e.g. a tendency to only use *draconian* to describe *punishment*).

```
0.0000000000 laud
0.0004700353 laudanum
0.0000000000 laudatory
0.0000000000 lauderdale
0.0000000000 laue
0.0307873090 laugh
0.0000000000 laughingstock
0.0000000000 laughlin
0.0126909518 laughter
0.0004700353 launch
```

Figure 4: Sample data from the representation of *joke*.

Despite the fact that "word A is within ten words word B" describes a symmetric relationship between words in a text, the word scanning software did not capitalize on this even though it would have cut database creation time in half. The symmetric relationship was neither used to halve the size of word representations in memory or on disk, though in this case a conscious consideration of the memory caching strategies of many modern CPUs (which make it preferable to iterate through contiguous memory locations rather than sparsely-distributed locations) led to the choice of storing the information redundantly.

Once all the word scanning was completed, the database consisted of thousands of files storing relatedness value information in ASCII text form. An excerpt from the representation of *joke* can be seen in Figure 4. Programs that use this information must endure the additional computational overhead of converting the ASCII text numbers into IEEE double precision floating point representations. Furthermore, listing the related word along with relatedness values is unnecessary, as this information can be divined from the position of the value in the file (since all words were related to the same list of dictionary words). Finally, storing the divided relatedness values is inconvenient to applications needing the original, undivided co-occurrence scores. These considerations led to the creation of a new, more efficient binary file format, where each word representation file was a one-dimensional array of undivided co-occurrence scores stored as unsigned long integers. Most of the programs that used the database loaded these files and divided their contents by the frequency of the represented word right away, but the option of having the raw scores was available to applications that needed it. In any case, the new database format eliminated several of the performance bottlenecks experienced by programs using the old format and allowed operations over the database (described in successive sections) to be performed more quickly.

Soon after programs that used the new database were developed, it became clear that the sheer size of the word representations presented the computer with a formidable computational task. With each representation consisting

of around 25,000 values and with over 6,000 words in the entire database, a simple comparison of one word with all the rest required the loading of 600 megabytes of data from disk and the execution of 300 million floating-point operations. This process took nearly seven minutes on a 300 megahertz Sun Microsystems Ultra 10 workstation. The only viable solution was to somehow reduce the size of the representations in order to gain speed.

A cursory examination of the ASCII word representation files revealed that many words co-occurred only very rarely or not at all, Indeed, the average number of nonzero relations among all of the represented words was 4,118. Many words appear so infrequently in the Project Gutenberg corpora that in most word representations they had a relatedness value of 0; some don't appear in the corpora at all. Based on these observations, it was decided that in paring down the word representations for speed's sake it made sense to eliminate these rare words first; relating at 0 all the time, they probably contributed little to the overall texture of the word representation. Two new databases were created in both ASCII and binary formats: one where word representations contained relatedness values corresponding to the 10,000 most frequently appearing words in the corpora and one with relatedness values corresponding to the 1,000 most frequently appearing words. The latter database resulted in an especially large reduction in processing time, mandating the loading of only 24 megabytes of data and the execution of only 12 million floating point operations. Accordingly, the same comparison task described above took only twenty seconds using the smaller database on the same hardware.

Motivated by slightly more academic interests, both Burgess and Lund and the LSA team have experimented with paring down their concept representation vectors. The HAL researchers claim that 100 to 200 of the principal components of their representation vectors are responsible for the better part of the distinction between them. Landauer, Foltz, and Laham describe significant benefits from reducing the dimensionality of LSA representations, though they don't specify in [16] how they do it. In any case, while it might be reasonable to assume that the principal components of the present word representations might well be relatedness values corresponding to the 1,000 most frequently occurring words in the corpus, no research has investigated whether this is the case.

2.4 Simple analysis techniques

With the databases created, it became necessary to see whether information about word meanings had actually been encoded and, if so, to what extent. Given the holistic nature of the word representations, it was impossible to analyze each representation relatedness value by relatedness value and determine whether it had "correctly" encoded the meaning of the represented word, as a correct representation could be dependent on conditions involving hundreds

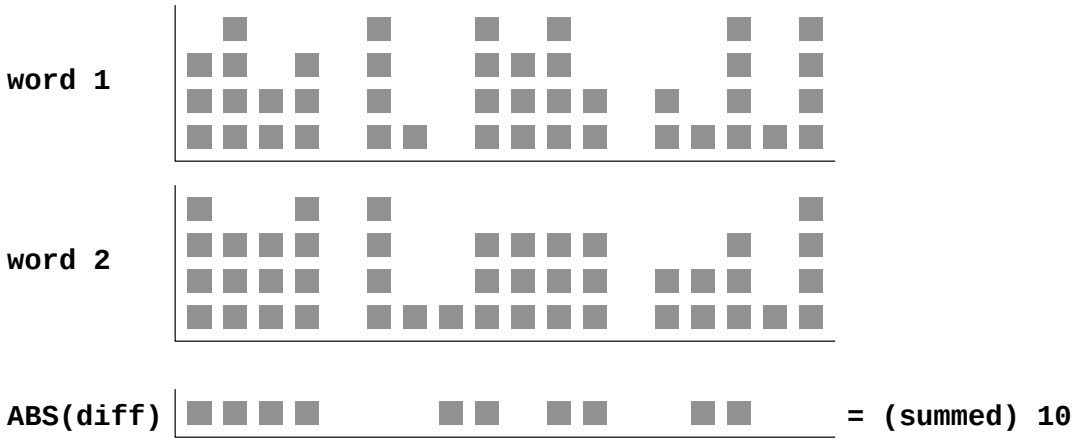

Figure 5: A comparison of corresponding relatedness values in two word representation vectors. These vectors are stylized versions of the real vectors used and only contain values in the range (0, 4).

of separate relations. Instead, perhaps the most straightforward method of analysis was to compare word representations against each other and see if they had semantic interrelations similar to those we distinguish between their corresponding concepts.

Perhaps the simplest of these interrelations to detect is synonymy. If the co-occurrence hypothesis is correct and each relation within a word representation consistently reflects some degree of semantic relatedness between the represented word and the related word (again, this relatedness need not itself be synonymy), then it might be expected that synonymous words, or at least words with very similar meanings, have a similar overall pattern of relatedness values. This overall similarity can be simply measured by computing the sum of the absolute values of the differences between corresponding relations in two relatedness values, as depicted in Figure 5. Indeed, this method does reveal that representation pairs corresponding to synonymous and related words do have more similar representations than unrelated words.

While it is possible to detect some hint of structure in the words database by comparing pair after pair of word representations, a more informative examination can be achieved by comparing one representation with all of the others in the database and seeing which ones are the nearest matches. A program was written to perform exactly this task, and consistently it revealed groupings of representations more or less in accordance with our own notions of what their corresponding words mean. Burgess and Lund and Rieger had similar insights and created roughly identical programs in the course of their work. Figure 6 contains examples of outputs from this project's nearest match program, with the twenty nearest matches arranged from worst to best. The first list is an example of better than average matching, the

second an example of average matching, and the third an example of poor matching, where "average matching" is a qualitative evaluation based on extensive use of the program. These lists of near-matching word representations don't show synonymy in the strict lexicographical sense. Nevertheless, most of the matches in the first two examples do have very similar meanings indeed. Finally, though almost none of the matches for the representation of *this* make any sense at all, the lack of real English words with similar meanings might excuse this result. Almost without exception, the more concrete the concept a word describes, the better the matches tend to be.

One adaptation of this program worthy of note allows an even more automated perusal of the database. Beginning with a word supplied by the user, it seeks the twenty closest-matching representations, chooses one at random, prints it, and then repeats the process with the new word. The output could be charitably described as a "train of thought" working its way through the database. An example of the program's output can be seen in Figure 7. This example demonstrates one of the notable features of this program: it tended to get caught in semantic "local minima", or oscillate for a while among words of similar meaning. Frequently-seen local minima included barnyard animals, names, adjectives meaning "bad", and kitchen utensils. This behavior suggested that, to a certain extent, representations of words with similar meaning occupy distinguishable clusters within the overall database, a suspicion confirmed by the cluster analysis described below.

It is worth noting that there are other means of comparing word representations besides the sum of the absolute values of the differences. Summing the squares of the differences is an extremely common technique, and though it was eschewed here ostensibly for the sake of speed (changing the sign bit of a floating point number is hypothetically faster than multiplying a number), tests of this method seemed to indicate no significant change in output or computation time. Another comparison method, the one used by the LSA team, takes advantage of the fact that the representations are essentially many-dimensional vectors and computes the cosine of the angle between two of them to return a comparison score between 1 (identical) and -1 (entirely dissimilar). Landauer, Foltz, and Laham make some claims about this method having some advantages over the other two [16]. The preferred method of the HAL researchers—finding the Euclidean distance between two representation vectors—is yet another option. These latter two were not tried.

In sum, simple analysis techniques revealed the presence of semantic structure within the automatically-generated words database. Compiling lists of near matches demonstrated that words of similar meanings, especially more concretely defined ones, have similar representations. The "train of thought" program, by looping repeatedly within groupings of high-similarity representations, hinted at the existence of distinguishable and perhaps semantically intu-

Better than average matching:

```
The 20 words most closely matching pennsylvania:
        germany:        (4.27233337309462)
        russia:         (4.26231411909167)
        scotland:       (4.2620328977226)
        ireland:        (4.25822822230981)
        texas:          (4.11167136334641)
        greece:         (4.10729752750801)
        philadelphia:   (4.04772470838356)
        northern:       (4.03810790626366)
        southern:       (4.01649418846413)
        palestine:      (4.00847457627116)
        carolina:       (3.98324020612156)
        spain:          (3.89657548830976)
        italy:          (3.87382020702563)
        california:     (3.83791471097109)
        boston:         (3.81784616909505)
        county:         (3.75225571911492)
        district:       (3.74691471073014)
        massachusetts:  (3.73639101011982)
        virginia:       (3.62903254062103)
        ohio:           (3.58961231820934)
```

Average matching:

```
The 20 words most closely matching red:
        star:       (4.05646500590785)
        straw:      (4.021607391558)
        marble:     (4.01380301254946)
        band:       (4.01094595329621)
        brown:      (3.98557618611866)
        gray:       (3.96593418845464)
        skin:       (3.95763561429687)
        tail:       (3.93012877430583)
        orange:     (3.81019198478384)
        violet:     (3.75253564671066)
        grey:       (3.68294679053349)
        green:      (3.6780977817684)
        pink:       (3.67644741973086)
        crimson:    (3.61129048798974)
        yellow:     (3.54211173604341)
        purple:     (3.53180275464562)
        scarlet:    (3.49892727774698)
        blue:       (3.49038476252189)
        black:      (3.25691697724813)
        white:      (3.07891081579041)
```

Below average matching

```
The 20 words most closely matching this:
        brief:      (4.3843867431262)
        hell:       (4.38047048370103)
        walter:     (4.37196305454726)
        sport:      (4.36566862313632)
        one:        (4.36461189972728)
        proof:      (4.35687726406875)
        it:         (4.35526084772066)
        correct:    (4.35291298966575)
        demand:     (4.34996690903279)
        charity:    (4.34785312490355)
        poison:     (4.31599627958627)
        cure:       (4.28106597971195)
        folly:      (4.25352692695508)
        today:      (4.22221930617487)
        murder:     (4.2187030355863)
        crime:      (4.21673555567411)
        grant:      (4.17901910928733)
        idle:       (4.14806000575788)
        for:        (4.09231898001682)
        that:       (3.99682256107578)
```

Figure 6: Better, average, and good results from the near-matching word representation finder.

```
vine->star->crystal->luminous->shade->gloom->blaze->flame->glow->shade->grove->
vale->fountain->lake->bay->sand->ice->sandy->hollow->shade->grove->vale->
torrent->waters->flood->tide->ice->mud->huge->sand->dust->shade->flame->fiery->
fierce->fiery->dust->fiery->dust->hollow->flat->ice->mud->bark->corn->milk->
soup->pot->tin->tar->nut->worm->snake->hare->cow->chicken->mouse->cow->hare->
pig->bill->smith->jack->harry->joe->harry->walter->robert->joseph->stephen->
jean->jones->jean->jones->jack->buck->hare->bull->lion->dragon->dwarf->monk->
hero->murder->blind->cruel->horrid->cruel->horrid->frightful->^C
```

Figure 7: Output from the “train of thought” program.

itable clusters within the data. These results seem to rely on a simultaneous application of the co-occurrence hypothesis and the holistic interpretation of word representations: while individual word proximity figures are prone to noise, such as words relating closely to *the* and very specific relations like *red* and *barn*, large representations consisting of many relations can be analyzed in their entirety and reveal similar meanings from different contexts. This is why *red* and *green* have very similar representations despite being used to describe different objects.

2.5 Cluster analysis

The results of the simple analyses called for a more in-depth look at the organization of the words representation database. A cluster analysis, which organizes vector data into a hierarchical tree of groupings based on a supplied distance metric, was an obvious choice. Indeed, Rieger presents a cluster analysis of his word representations in [14] and uses it to demonstrate semantically appealing clumping within his data.

Many sophisticated algorithms exist for clustering, but some of them require the creation of adjacency matrices or other data structures that, given the number of elements in the word representation database, would have been impractical or impossible to use on available hardware. Meanwhile, several advanced third-party clustering applications would have required the conversion of the word frequency database into a specified input format. In the end, a very simple, slow algorithm was used: it merely scanned the database for the two closest matching representations, extracted them, connected their corresponding branches in the cluster tree under construction, and replaced them in the database with a single new new representation created by taking their average. The distance metric used was the same sum of absolute values metric used in the simple analyses, and the database used was the 10,000 relation per word database. The resultant cluster tree, depicted graphically, can be seen in Figure 8.

While this tree is too intricate for detail to be easily perceived, special indicators have been added to highlight particular areas of structure. These were manually added to clusters after the fact, and while they reflect notable

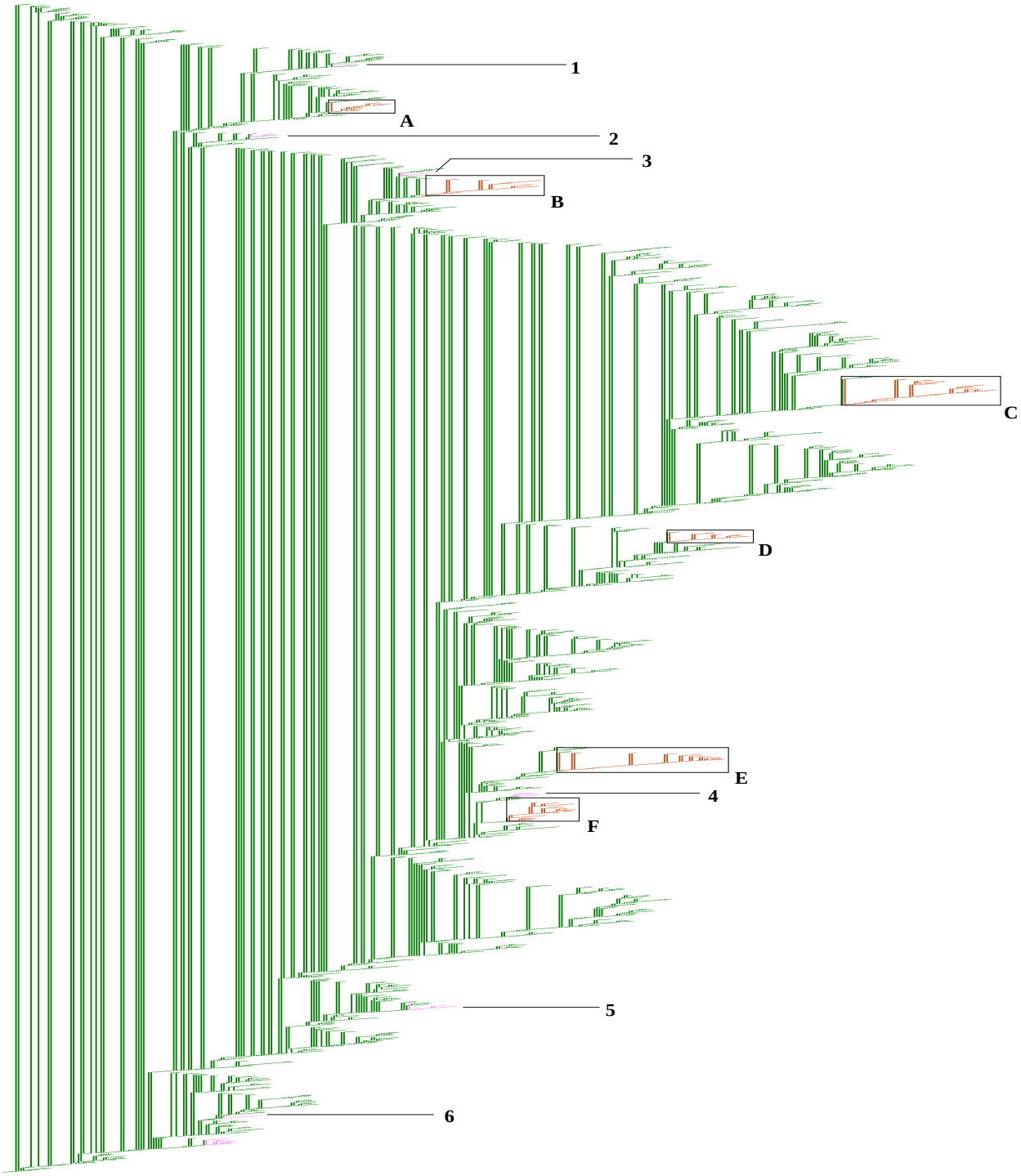

Figure 8: The large cluster analysis tree, with over 6,000 individual leaves. Indicated portions correspond to specific clusters identified in Table 1; features identified by numerals are too small to box.

Small features

1. Nine of the twelve months
2. Geometric and measurement notions (e.g. triangle, breadth)
3. Several old-fashioned seeming words (e.g. boon, fain, hither)
4. Foodstuffs and food ingredients (e.g. flour, vinegar)
5. Number words (e.g. forty, nineteen, three)
6. All days of the week and other day words (e.g. christmas)

Larger features

A. Words relating to departure
B. Mostly negative interpersonal action words
C. Many human qualities and characteristics
D. Basic non-interpersonal action words
E. Animals and dry goods
F. Names and nationalities

Table 1: A legend for the indicated portions of the cluster tree. Small features correspond to more specific groups of concepts and to smaller portions of the tree. Larger features, appropriately, correspond to larger tree portions and broader conceptual groups.

groupings found during a perusal of a 25,000 line data file, it should not be assumed that they are atypically good or bad examples of organization within the database. The areas highlighted by the indicators are described in Table 1.

Other notable groupings include intentional verbs (e.g. *believe*, *think*, and *intend*), names of characters from Shakespeare's plays, terms from geography and transportation, and more. Typically, word clusters displayed at the top of the cluster tree are more coherently grouped than those at the bottom, where for example *asterisk*, *tilde*, *deductible*, *champaign*, *newsletter*, and *literacy* are close neighbors to each other. This is an understandable result. Pairings made at the very beginning of the clustering process formed the first leaves—later, as more clusters were created from their averaged representations, the tree grew beneath them. The leaf pairs near the bottom of the tree were not optimal pairings until most of the rest of the tree had already been created. This "choosing from leftovers" helps explain why certain pairings appear to be somewhat tenuous.

The cluster analysis reveals that the internal structure alluded to by the simple analysis techniques of the previous section does exist. However, it seems equally evident that this structure is not strict or pervasive, at least not in a way that is intuitable or significant to human beings. Aside from smaller groupings (e.g. days of the week), clusters can contain words from different parts of speech and different basic lexicographic categories. This means that it would be difficult to derive useful information about a word strictly from its neighborhood in multidimensional space. The representation for *believe* is similar to the representations of other propositional attitudes. However, its slightly more distant neighbors include *why*, *sure*, and *grudge*. No broad attributes about *believe*, such as whether it's a verb, whether it's an observable action, whether it's transitive, and so on is apparent from other nearby words.

2.6 Neural network analysis

Even if the semantic space of the words database was organized more coherently, simple location probably wouldn't be enough to distinguish important categorical information. Concepts can belong to many different categories—a sledgehammer is a tool, a thing containing metal, a heavy thing, an oblong thing, and so on.

Just because it would be extremely difficult to organize a semantic space where vicinity reflects most of this categorical information doesn't mean that category membership can't be deduced from distributed representations. It may be possible to discover patterns within a representation's relations that encode all kinds of information. One way that shows some promise is to train neural networks to distinguish these patterns. The considerable size of the representations in the words database do require neural networks with correspondingly large numbers of input nodes and node interconnections, a potential drawback. Regardless, with patience, it appears that neural networks can be made to detect categorical and other features within word representations. It should be noted that this work took place near the end of the Summer 2000 research period, and thus there are considerable avenues here that have been left unexplored.

Neural networks offer a wide assortment of design choices. Since the present research focused more on using them as a tool, the neural networks employed were all quite familiar and generic—simple two-layer feed-forward networks. In all cases, the activation function was a simple sigmoidal function. All training was done with straightforward backpropagation.

Applying neural networks to the words database required a considerable amount of custom code. Free and commercial neural network packages abound; however, most of them are designed to take in input in the form of text data files, many aren't designed with thousands of internode connections in mind, and many others are packed with unnecessary features (e.g. visualization tools, diverse training algorithms, support for various network architectures, and so on). Thus, to enhance speed and ease of use, a series of C++ programs were created to train and execute the neural networks. Their chief asset was that they permitted the user to simply specify the names of word representations used as network input and training data—a more general purpose program would have required the conversion of these data into one or several very large input files. It is worth mentioning that these C++ programs were not entirely homemade but instead were mostly a set of wrappers around version 1.0.2 of Daniel Franklin's free libneural library[6], a C++ library implementing two-layer neural networks and simple backpropagation training.

The first application of the neural network software was category recognition, a good choice for several reasons.

[6] `http://ieee.uow.edu.au/~daniel/software/libneural/`

Figure 9: Meats.

First, as mentioned earlier, it couldn't be done with the more conventional methods explored thus far. Second, most category detection applications could do with a single output node offering an affirmative or negative answer to whether a word belonged to a particular category. Even though the neural network software had been custom tailored to deal with word representations, the opportunity to reduce the number of network nodes is valuable due to the vast number of interconnections between nodes and hence the vast amount of computation necessitated by backpropagation. A network with a 1,000 node input layer (for a single word representation), ten node hidden layer, and single node output layer has 10,010 trainable connections. Even for simple networks like these, extensive training remains a challenge.

The first category discerning networks were trained to discern whether an input word was a member of a particular category. The first category was meats (Figure 9)—a relatively straightforward grouping if ever there was one. This did not mean it was without problems. Despite the over 6,000 words within the database, perhaps fewer than a hundred had something to do with meats. Of these, many were tenuous choices. An unfortunate consequence of relying exclusively on the co-occurrence hypothesis to generate word representations is that multiple senses of a word are bound together in the same representation. One of the words used as a positive example of a meat in neural network training was *turkey*—though most runs of the network indeed recognized *turkey* as a meat, it's not unlikely that being trained to recognize words with non-meat-related senses caused the network some difficulty in acquiring meat's essence. It may be safe to conclude that for most concrete categories of any specificity, the problem of having to choose between a very small training set of clear members and a larger training set including some tenuous members will always be

Network	Hidden nodes	Maximum error	Missed unique nouns	Missed unique non-nouns	Total missed unique items
isanoun1	10	0.0025	153 (7.9%)	119 (8.3%)	272 (8.1%)
isanoun2	5	0.001	158 (8.2%)	114 (8.0%)	228 (6.8%)
isanoun3	3	0.005	152 (7.8%)	127 (8.9%)	279 (8.3%)
isanoun4	2	0.005	167 (8.6%)	130 (9.1%)	297 (8.8%)
isanoun5	1	0.005	n/a	n/a	n/a
isanoun6	2	0.001	228 (11.8%)	101 (7.1%)	329 (9.8%)
isanoun7	25	0.005	156 (8.1%)	120 (8.4%)	276 (8.2%)

Table 2: Configuration and training of the noun-recognizing networks. All networks were trained for 200 iterations at a learning rate of 0.05, then made to categorize all the words in the database. (The isanoun5 network could not successfully be trained.) The results cataloged above concern only words which were definitely nouns or non-nouns (i.e. not a word like *wish*, which is a noun and a verb). There were 1,931 of the former and 1,432 of the latter, for a total of 3,363.

present.

Around twenty varying neural network configurations were tried. None were totally successful—typically for every meat word judged "meaty" by the network, perhaps five non-meat words would also be included. This may seem like a rather inauspicious result, but again the nature of the training data must be taken into account. One of the better networks, featuring a twenty five-node hidden layer, accepted 292 of the six thousand words as meats (i.e. for these words, the output node bore a value greater than 0.5), of which 60 or so were actually legitimate choices. Details on the training and output of this network can be found in Figure 10. Among the more unusual aspects of this network is the very small number of positive meat examples: only 19 in total, chosen for their relative unambiguity as meats. In retrospect, experiments with more positive examples may have been appropriate. Overall, it's not clear what factors—limited positive examples, usage regularities, word ambiguity, and so on—lead to the inclusion of the over two hundred spurious meat words in the results. Regardless, the words deemed "meaty" by the network range from very frequently used terms to obscure ones indeed, making it unlikely that the problem has to do strictly with obscurities relating to usage.

Some categories are easier to learn. The second and last specific category attempted was nouns, and here the network found greater success. Using information from the WordNet database, words in the words database were identified as nouns and non-nouns. Words like *envy* fit both categories, so these sets were pared further until they consisted of 1,931 unique nouns and 1,432 unique non-nouns. 500 random choices from both sets were placed into the training corpus, which was used to train neural networks of identical architecture to the meat networks. Networks with varying numbers of hidden units and were trained with varying maximum mean-squared error values (see Table

Configuration

Number of input layer nodes:	1000
Number of hidden layer nodes:	25
Number of output layer nodes:	1

Training

The nineteen examples of meat words in the training corpus were: *bacon*, *beef*, *buffalo*, *chicken*, *dove*, *duck*, *fish*, *fowl*, *goose*, *hare*, *hen*, *lamb*, *mutton*, *pork*, *poultry*, *rabbit*, *meat*, *tongue*, and *turkey*.

There were 462 words presented as non-meat words.

The training data specified an output node value of 0.01 for example non-meat words and an output node value of 0.99 for meat words.

Training was simple backpropagation. The training examples were presented to the network in a new random order on each iteration.

Iterations over training examples:	500
Learning rate:	0.5
Maximum mean squared error:	0.007

Results

The network identified 292 of 6,336 words as meats. 60 of these appear to be legitimate choices. A few obvious meat words from the database (e.g. *cow*) are missing.

Legitimate meats recognized as meats:

Word	Score	Word	Score	Word	Score	Word	Score	Word	Score
hound	0.527757	steer	0.745365	turkey	0.834475	deer	0.942585	salmon	0.984208
livestock	0.585588	cock	0.747737	stag	0.845899	mutton	0.943656	duck	0.986126
peacock	0.602632	whale	0.757616	sheep	0.860356	meat	0.952071	dog	0.987362
bone	0.605993	squirrel	0.762431	calf	0.863865	hen	0.956942	fowl	0.988638
bass	0.625897	turtle	0.772783	carp	0.901693	swine	0.961653	ram	0.98864
bull	0.644903	shark	0.781112	hare	0.925178	cod	0.963283	roast	0.991264
boar	0.650033	swan	0.788557	goose	0.925546	poultry	0.96565	bear	0.997868
oyster	0.71308	goat	0.797847	dove	0.930371	ox	0.967269	tongue	0.998265
trout	0.727683	antelope	0.798683	lamb	0.938802	chicken	0.969539	flesh	0.998458
hog	0.731514	cattle	0.808652	colt	0.939827	pork	0.973189	bird	0.9995
bacon	0.736676	oxen	0.815601	rabbit	0.940119	beef	0.980789	game	0.999894
pig	0.740467	buffalo	0.817282	bullock	0.941837	sow	0.983059	fish	1

Sampling of 60 non-meats recognized as meats (there are too many to fit them all):

Word	Score	Word	Score	Word	Score	Word	Score	Word	Score
dryden	0.500066	arrange	0.64102	allied	0.78227	dairy	0.938433	contract	0.983039
chess	0.511804	cabbage	0.647638	food	0.79443	breach	0.944873	newsletter	0.989603
bind	0.521059	they	0.661254	blackstone	0.804069	verb	0.951974	explore	0.992443
contribute	0.52811	cable	0.675215	behave	0.826922	imitate	0.958993	designate	0.995295
indebted	0.538954	dan	0.691485	confine	0.8462	additional	0.962218	substitute	0.997146
bilateral	0.541946	come	0.720331	tackle	0.867653	stroll	0.969149	assign	0.997923
cling	0.56534	fuel	0.728507	join	0.880479	bring	0.969756	assemble	0.998406
fasten	0.573063	soon	0.730187	convey	0.882367	appoint	0.971401	identify	0.999292
extend	0.5843	the	0.74668	gray	0.889054	bin	0.973983	non	0.999688
good	0.595525	conform	0.750972	build	0.898645	plunder	0.977062	ourselves	0.999922
toll	0.614069	instruct	0.755079	potatoes	0.908704	dig	0.979947	ex	0.999996
angeles	0.632045	drain	0.770369	bathe	0.913986	administer	0.981142	eat	1

Figure 10: Information on Meat22, a relatively successful meat network.

2). The progression of training parameters describes a period of experimentation in which it was determined exactly how many hidden layer nodes were necessary to detect nounship. Two appears to be the number—a network with one hidden node, effectively a perceptron, was impossible to train for this task.

Once trained, the networks were made to judge whether each word in the entire database was a noun or a non-noun. The results are a bit tricky to interpret, as a judgment on a word like *envy* could be correct either way, or could depend on whether its noun sense or verb sense is more commonly used. Rather than find usage data for the ambiguous words, it was decided that the network's performance on unique nouns and unique non-nouns would suffice. As Table 2 demonstrates, the neural networks typically misidentified just over 150 unique nouns as non-nouns and around 110 unique non-nouns as nouns. In comparison with the meat networks, these networks are considerably more successful. What's also interesting is the remarkably low number of hidden nodes needed to make the distinction. Even though nouns and verbs intermixed with some regularity within clusters generated by the cluster analysis, their representations evidently bear easily identifiable features that can be detected by neural nets.

Having learned that neural networks could detect at least certain categorical features, an ambitious effort was undertaken to see if neural networks could generalize the notion of category membership. For this, a neural network would have to take in two words and determine whether the first word was a hyponym of the second. These efforts were largely unsuccessful, but the experimentation performed was limited by available time—it was not conclusively established whether this kind of processing is possible.

Two networks were constructed, both with 2,000 input nodes (a separate thousand for each word) and a single output node. The first network had a hidden layer of fifty nodes, while the second had a hidden layer of 200. The training data was exceptionally difficult to create, requiring the manual creation of hyponym/hypernym pairs from words in the database. As such, only a limited number of positive categorical matches were made. The generation of negative categorical matches was speeded by a Perl script which randomly generated word pairs and asked the user to judge whether they were an example of categorical inclusion; in all but one of 254 iterations, the answer was no. In sum, both networks had 95 positive examples of hyponym/hypernym pairs and 375 negative examples.

Both networks were tested against a selection of positive examples from the training corpus and some 20,000 randomly-generated word pairings. Neither appeared to have achieved the general notion of category membership. They failed to detect all but of a handful of the positive hyponym/hypernym pairs from the training set and spuriously identified hundreds of others as demonstrating this relationship. Nevertheless, this does not conclusively establish whether neural networks are capable of performing this task. With over 40 million possible word pairings from

the database, it's not unlikely that the size of the training corpus was simply inadequate. Future research into this area might benefit from extracting better training data from WordNet or some other compilation of lexicographical information.

Thus, while it is clearly possible to use neural networks to deduce some kinds of categorical information, it's not clear yet what kind and how much. It's easy to attribute the mediocre performance of the meat detecting network to an insufficiently sized training set, especially in light of what the noun detecting network could do. From a strictly practical standpoint, however, this calls the utility of neural networks into question—if there are relatively few examples of a detectable category to begin with, it's difficult to conceive of how (or why) a neural network would be trained to detect its members. In any case, more research is necessary to discover the utility of applying neural networks to massively interlinked word representations.

2.7 Word representation conclusions

The experiments presented here offer an overview of some of the present capabilities of two interesting techniques: massively interlinked graph structures and the co-occurrence hypothesis. When used together at least in domains like word meanings, where multitudes of distinct elements of the same type exist in a space where proximity can hint at interrelationships, they evidently create large, semi-structured representations of meaning "on the cheap," or without much interference on behalf of the experimenter.

It's clear that the experimentation described here fails to comprehensively detail the capabilities of these representations. There are many potential avenues of investigation to be taken still, some of which may lead to gainful practical applications of the techniques:

- The neural network work could be greatly expanded upon, especially with further help from lexicographical databases like WordNet. There are many more categorical and grammatical distinctions that a neural network might be able to make. What's more, it might be possible for very large neural networks to directly create or modify word representations themselves.

- It would also be interesting to determine whether the word representations described here could be combined in methods similar to the fuzzy union, intersection, and difference operations described by Rieger in [13].

- On a much more basic level, the creation and configuration of word representations shows room for improvement: Burgess and Lund's work suggests that 1,000-relation word representations might be five times too large,

to say nothing of 10,000-relation representations or complete word representation graphs. Since the word representations used here are essentially the same as HAL word representations, it is likely that smaller ones consisting of the principal components of larger ones would demonstrate better performance. However, it would also be interesting to determine whether this size preference reflects a fundamental characteristic of massively interlinked semantic structures or is a byproduct of the co-occurrence hypothesis.

- It might be possible to detect two or more different contexts in which a single word appears and use this information to detect and separate multiple senses of the same word.

These and other considerations, as well as other research and application work in the field, suggest that there may be considerable benefit in using massively interlinked representations in language processing. Search engines, for example, are a present and expanding application: Latent Semantic Indexing (LSI) uses LSA to generate descriptive representations of texts that can be used for concept-based searching. How generally applicable, though, are massively interlinked representations? Are they applicable to other kinds of information representation? The next section details preliminary steps toward answering this question, describing an initial effort to apply them to environment and task learning in robotics.

3 Robot control

Summer 2001 brought a dramatic shift in the ongoing investigation into the applications of associational computing. The techniques developed the prior year for the purposes of describing abstract, relatively unchanging concepts would now be applied to the very dynamic world of robotics. This may seem a rather ungainly leap, but in fact it was a planned, if ambitious, extension from the more successful aspects of the prior year's research.

The goal was to develop a basic mechanism through which a robot could be trained by a human operator to perform a simple task. During training, the robot would record the movements and environment of the task and use the data to construct a large, well-connected graph data structure similar to (but far smaller than) the word representation graphs developed previously. Once training was completed, the robot would continually maintain information about its current sensory input and prior actions and use this information to derive appropriate actions from information stored in the graph. This scheme might appear at first glance to be little more than straightforward instance-based learning, in which all inputs and outputs to the system are merely recorded and later recalled by a program that tries to match current sensory input to a stored stimulus/response pair. There is, in fact, more substance to it. In constructing the

graph, the robot control program is attempting to make analogies between situations by creating structurally similar representations. Ideally, nodes that together represent a certain portion of a task, e.g. dodging an obstacle, cluster in many-dimensional space. This gives the robot a chance of being able to interpolate actions when it encounters unfamiliar stimuli.

In the course of research into this scheme, a somewhat successful attempt was made at training a robot to perform a simple, stateless task. Some progress was made in getting the robot to perform a stateful task, but thus far efforts to prevent the robot from losing track of its state have proven unsuccessful.

3.1 Related work

Using massively interlinked structures to represent concepts is not as common a practice in robotics as it is in other fields. This section describes other work that provides useful background information for the techniques explored here or that explores similar ideas.

Perhaps the simplest application of distributed representations toward robot control is instance based learning, described briefly in the prior section. Andrew W. Moore and Kan Deng offer a concise, informative analysis of this technique in their 1995 paper [17]. They describe how to use kernel regression to derive new motor outputs from stored samples of sensor inputs and actuator outputs. Here, a selection of samples with sensor input values resembling the present sensor data are collected, and the outputs of these samples are averaged together using a weighting technique that favors the matches whose sensor values were the closest match in the search. This technique is more advanced than the simple averaging used in the experiments described below.

The most interesting topic Moore and Deng discuss is how to reduce the time it takes to locate near matches of multidimensional objects in large collections of them. The technique used in the experiments below was simple: it merely compared a single representation to all the other representations and kept a record of near matches. Moore and Deng describe a storage technique called *k*d-trees in which a tree recursively subdivides multidimensional space into smaller and smaller blocks. Finding similar nodes is as easy as locating a branch or branches of the graph that describes a small enough area in multidimensional space. In doing this, it is possible to dismiss entire regions of multidimensional space which clearly contain dissimilar representations without checking each one of the representations individually.

A more complex use of distributed representations in robotics (if they can be called that) is the application of neural networks. This is a wide research field with many areas of active investigation, but a particularly good example can

be found in a 1993 paper by Meeden, McGraw, and Blank [18]. This example is particularly helpful because in this research, a robot controlled by a neural network had to learn to keep state and orient itself around a particular space, and some of the experiments with the present learning mechanism had similar goals.

The robot described in this paper was trained to repeatedly approach and withdraw from a lamp such that light sensor readings reached certain prescribed thresholds. The robot was an eminently non-holonomic toy car, so in addition to the overall task, it had to learn how to steer itself. It also had to be capable of avoiding walls. A two-layer recurrent neural network was employed, with state maintained by the feedback of activation through "backwards" connections between layers (i.e. connections from nodes in the hidden and output layers to nodes in the input layer). Training was done not with conventional backpropagation but with a technique called complementary reinforcement back-propagation (CRBP) in which the neural network is trained not by correcting for specific desired outputs to certain inputs, but by a more general system of rewarding and penalizing desirable and undesirable outputs respectively. Since the neural network "representation" of this task and the training method are quite unlike any used in the present experiments, there is no need for further detail on this work; suffice it to say that distributed representation in robotics finds vigorous (if tenuously defined) application in neural network research.

Besides the overall use of distributed representations, the analogy-making described in the introduction to this chapter is another very important problem in artificial intelligence and robotics. An interesting Ph.D. dissertation by Tim Oates focuses directly on this issue, and does it at the crossroads of the two subjects explored by this paper—word representation and representations of robot tasks [19]. Two experiments are described in this paper. In the first, a computer enabled with audio and visual inputs had to analyze a speech stream describing an arrangement of objects in a simultaneous series of images, segment it into words, and determine which words referred to which objects, colors, or spatial arrangements in the pictures. In the second, a robot analyzed its own video input and actuator outputs and inferred cause and effect relationships between motive commands and movement in the environment as seen on the camera. Both tasks were performed with the help of a general algorithm that analyzed streams of data and looked for repeated patterns of actions or values therein. As best it could, it isolated these patterns into discrete segments. It then used statistical techniques to detect correlations between co-occurrent streams, building up evidence that a certain sound was correlated with a certain shape, or a certain motor movement was correlated with a certain on-camera motion.

3.2 Contrasts and challenges

As mentioned, representing words and representing a robot's task and environment are very different things. Indeed, the contrasts between these two tasks motivated many of the architectural choices that characterize the learning mechanism developed during Summer 2002. What follows are descriptions of the most difficult differences and accounts of the design decisions they inspired.

3.2.1 Robot task and environment data are continuous

Within the words database, the concepts to be represented were discrete units. The words that referred to them were easily isolated in texts, allowing the straightforward distillation of contextual information. The real world is not so accommodating—it's not clear how to segment experience into meaningful chunks. One approach to this problem is simply not to bother trying. Instead, the robot control program generates new nodes at frequent intervals, much like a basic digital audio recording samples input from a microphone. It's up to the program to try and relate like samples to each other, and, in doing so, generate clusters of samples that may or may not correspond to discrete events.

3.2.2 No foreknowledge of concepts to be represented

A related problem concerns the fact that, unlike with the words database, it is not apparent exactly which concepts need to be included to cause the effective representation of a particular task. It is neither clear that it could be apparent with further analysis—robots interact with the world far differently than humans do, and our intuitions on how the robot should interpret the world are not necessarily correct. For this reason, the robot must develop its graph structure on its own—itself another motivation for the simple "sampling" approach to node creation. There are further ramifications, however. Naturally it is important that the graph not continue to grow forever, so there must be a limiting mechanism that determines when the task is sufficiently represented and ceases the production of new nodes. This latter problem was not directly approached in the course of the Summer 2001 research, though some possible solutions are offered below.

3.2.3 Maintaining thorough linking

With the graph structure being built up node by node, and without a foreknowledge of what the nodes will end up representing, it's difficult for nodes created early on to contain relations to nodes created later. Since the effectiveness of

concept representation with massively interlinked structures appears to depend rather heavily on massive interlinkedness, it appears important that older nodes become linked to newer nodes as the construction of the graph progresses. The Summer 2001 research confronted this problem, though not entirely successfully, by using routines that updated older nodes and by relying on newer nodes to assume the roles of older ones.

As a consequence of this sometimes-inconsistent linking between nodes, some nodes contained more outbound relations than others. This made it more difficult to consider each node as a point in multidimensional space, standing in contrast to the word representation research.

3.2.4 Multiple types of information

The words database consisted of only one type of information: word meanings. For a robot to function effectively, it must manage multiple types of data: information about its environment, about its task, about how to maneuver, information from its sensors, and so on. We wish the robot to automatically acquire the first three types of information with the help of massively interlinked structures. Should three distinct structures be used or one that somehow combines these types?

The Summer 2002 research took the approach of combining all the information into a single database. It did so first by establishing a set of nodes in which each node corresponded to a single robot sensor input or actuator output. These nodes contained no outbound edges of their own. Later, when the robot control program created new nodes from samples of the robot's current activity, they included relations to these nodes with relatedness values corresponding to the activations of their respective sensors or actuators. Finally, when the control program derived new nodes from the database, it merely had to seek out the relations corresponding to the actuators to execute the new action. Figure 11 presents this approach graphically.

These combined nodes don't seem to encode any information about the environment or task of the robot. In fact, this is handled by the self-clustering capacity of the network. Nodes corresponding to similar portions of a task and bearing information about a certain aspect of the environment do so by being similar to other nodes that do the same thing. This "dual-role" storage of information is not new. Neural networks that guide robots can store both environmental and task information within their connection weights—as an example, see the Meeden, McGraw, and Blank paper described earlier [18]. Even straightforward instance-based learning uses the same strategy, with the implicit assumption that inputs and outputs will differ when the robot is in different situations.

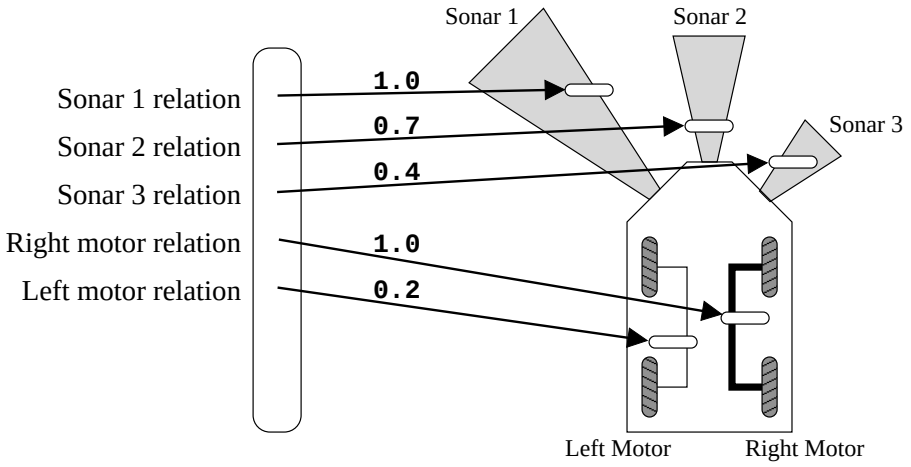

Figure 11: Nodes represent the state of the robot's sensors and actuators with relations to specially-created relationless nodes, of which there is one for each sensory input and actuator output. The relatedness values are proportional to the strength of their relation's corresponding sensor reading or activation value. Above, the stylized robot is turning to the left, hence the higher relatedness value for the right motor relation.

3.3 Architecture and operation

With these design decisions established, it is possible to step back and examine the entire system, which is documented graphically in Figure 12.

The robot control program consists of a store of previously created nodes (the graph structure) and a "current node" or "accumulator node": the node that is currently being created by the program in the course of training or autonomous operation. All operation of the control program is characterized by the construction of a new current node from sensor information and existing information within the graph.

The control program operates in one of two modes: training mode and execution mode. In training mode, the robot samples the inputs and outputs at regular intervals and embeds them in the current node as relations to the sensor and actuator nodes described in the previous section. Next, it adds additional relations to the current node in order to make it similar to other nodes that describe similar events. The precise nature of this addition was a subject that received considerable investigation over the course of this research and naturally wasn't infallible in any of its forms. The strategies used to add more relations included ones which simply added relations to nodes with similar inputs and outputs, ones which added relations added to recently created nodes, and others. (These will be described more thoroughly in the next section.) Once the current node is created, it is either added to the graph straightaway or, depending on the preference of the trainer, only if it is sufficiently different from other nodes already stored in the

The initial configuration of the robot control system. Nodes representing past experience are kept in the node storage database. Besides their interrelations, these nodes also contain relations whose relatedness values represent the sensor inputs and actuator outputs of the robot at the time of their creation. Meanwhile, the accumulator node waits to be filled with new information.

The control system gains experience in training mode. Information about the current state of the robot's sensors and actuators is stored in the accumulator node as relations to a few specially−created "a priori" nodes (five in these images). Further relations describing the immediate circumstances of this node's creation can be added next (not shown). Finally, the accumulator node is imbued with relations to other nodes within node storage, typically based on how closely it resembles these nodes. The goal of these manipulations is for the nodes to automatically form clusters representing related concepts within node storage.

In execution mode, the control system must derive its actions based on sensory input and prior experience. It starts by placing relations describing sensor input into the accumulator node. Further relations describing the immediate circumstances (e.g. nodes created or accessed recently) may be added next (not shown). The accumulator node is next imbued with relations to other nodes in node storage, typically based on how closely it resembles these nodes. It then is compared again with stored nodes and uses the actuator relations from the nearest matches to compute the relatedness values of its own actuator relations as well as the actual actuator settings for the robot.

Figure 12: A schematic diagram describing the learning and execution processes of the robot control system.

graph. This latter mode of operation exists to prevent the graph from storing information it has already sufficiently learned.

In execution mode, the robot is acting completely autonomously. Here, the control program samples only sensory input and adds the corresponding relations to the current node. It then adds additional relations using the same process as the one used in the training phase. Finally, it searches the database for similar nodes. A specified number of top matches are collected and their outputs are used to derive the appropriate action. This derivation can be as straightforward as simply averaging the outputs or something more complicated, such as using linear regression to determine the value of a particular actuator-related relation. Here, the top matches are arranged on an axis according to how well they matched the current node. (Node comparison is described in the next section.) The matches have comparison scores ranging from 0 to 1, with 0 as the least related and 1 as the most related. Thus the comparison score can be thought of as the independent variable and the relatedness values of actuator relations can be thought of as dependent variables. A regression line is computed for each actuator variable, and the value of each line at the comparison score of 1 (a hypothetical perfect match to the current node) becomes the relatedness value of the corresponding actuator relation in the current node. If this overshoots 1 or undershoots 0, it is set to the corresponding extreme bound of this interval. For a more comprehensible graphical representation of this process, see Figure 13.

Once the current node is fully constructed, the relatedness values on the actuator relations are extracted and turned into actual motor commands. Again, depending of the preference of the trainer, the node can then be discarded entirely, inserted directly into the graph, or inserted only if it is deemed sufficiently different from already existing nodes.

The scheme described thus far permits nodes to have any number of relations, limited only by the size of the overall graph. Since this threatens the computational tractability of the system, the robot control program typically limits the number of relations the nodes could bear to a specified value. At present, it selects that many of the highest-valued relations. The ramifications of this design choice are discussed in the next section.

In the course of operation, the trainer can switch the robot back and forth between training mode and execution mode at will. There is no means of specifying beforehand how long and when the training and execution phases will take place.

3.4 Comparison operations and node construction

Continuing into the particulars of this scheme, we are now able to examine the way nodes are compared to each other and how these comparisons are used create and maintain nodes—the current node, and occasionally nodes already

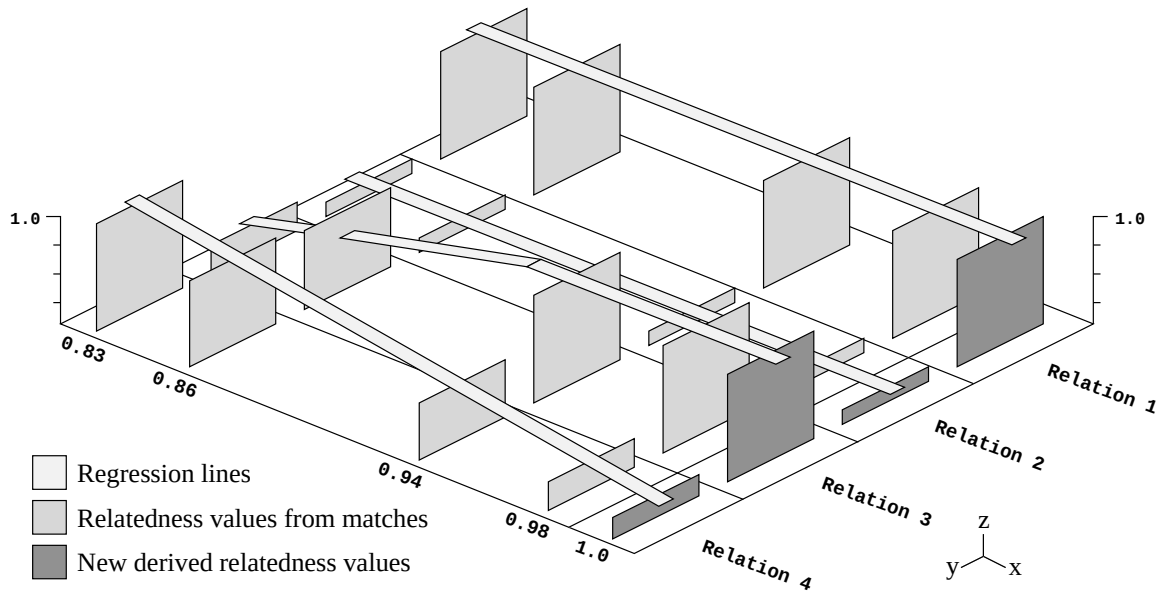

Figure 13: A graphical depiction of using linear regression to derive new relatedness values for actuator relations. The four nodes closest to the target node have been arranged along the x access according to their comparison score. Each item along the y axis represents a different relation, with relatedness values occupying the z axis. For each relation, we use linear regression to find the value of z at $x = 1$, a perfect comparison score. Intuitively, this process looks for trends in the database, determining how actuator values change as stored situations become more like the present situation. Note, finally, how regression lines are clipped at $z = 1$.

inserted into the graph.

As mentioned earlier, it is likely that two nodes to be compared feature different sets of outbound relations. This means that a simple vector comparison like the one used in the word representation research is not possible, or at least not as easy. Several strategies exist for comparing nodes, but they can all be divided into two groups: those that don't attempt to find out what missing relatedness values are and those that do. The first may produce less thorough results than the second, whose members are too slow to be practical for real-time robotics.

The first group of comparisons are known as base comparisons. These all work by averaging the differences in like relatedness values in two nodes, then subtracting this average from 1. Since relations range from 0 to 1, a comparison score of one thus indicates that the two nodes are identical, while a score of zero indicates that they are as different as they can be. The procedural differences within this group arise when relations which aren't common to both nodes are encountered. There are four present approaches:

- **LPCBase1**: Computes the comparison score by using only the relations the two nodes have in common. If they have no relations in common, it returns 0.
- **LPCBase2**: In calculating the average, it sums the differences in relatedness values of only common relations. It divides, however, by the total number of distinct relations in both nodes, effectively giving relations unique to only one of the nodes a difference value of 0. This is tantamount to assuming that nodes are identical when one can't tell otherwise, and thus it artificially inflates comparison scores. For this reason, this comparison is seldom used.
- **LPCBase3**: This comparison sums up the differences in all the relatedness values seen in both nodes. If a relation is unique to just one node, it assumes a relation of value 0 in the other. To compute the average, it divides by the total number of distinct relations in both nodes. This comparison is the most thorough of all the base comparisons and is most frequently used.
- **LPCBase4**: This comparison is identical to LPCBase3 except that it assumes a relation of value 1 when a node is missing a relation. This comparison has little utility.

The second group of comparisons all attempt to discover what the missing relatedness values should be before computing the average. These are all recursive functions that, upon encountering a node with a missing relation, determine the relatedness value for that relation by comparing the node to the related node on the spot. Because this is yet another comparison, it's possible for the process to repeat the comparison again *ad infinitum* unless some limit is put on the recursion. The two comparisons **LPCRecursive** and **LPCRecursiveAdd** require the program to specify the recursion depth and a second comparison to be applied when the comparison has recursed that deeply. Typically this comparison will be one of the base comparisons listed above, but any comparison may be used.

LPCRecursiveAdd only differs from **LPCRecursive** in that after calculating the value of a missing relation, it adds it to the node it was originally comparing. This removes the need for further recursion on future comparisons, though it does not account for the possibility of a related node changing after the creation of the relation. If the related node changes dramatically, the relatedness value might not accurately reflect the difference between both nodes.

These comparison mechanisms are crucial to the construction of the current node in both the training and execution phases. They are used, in brief, to force clustering among the nodes in the database. There are, once again, several possible choices for how this is done. What follows is a step-by-step description of how the most common model operated.

Recall that the creation of the current node first involves the insertion of relations to sensor nodes and, if the control program is in training mode, actuator nodes as well. At this point the control program must imbue the current node with further relations in order to make it similar to other nodes that represent similar notions.

Because the robot control program is continually sampling actuator and/or sensor data, and because the sampling rate is relatively rapid, we can assume that the situation that will be described by the current node will not be drastically different from that described by the few nodes created immediately prior to it. Some information from immediate context can thus be inserted into the node. In the experiments conducted thus far, this is done by averaging the like non-sensor and non-actuator relations in recently created nodes and inserting them into the current node. A relatedness value of 0 is assumed when a relation is missing in one of the nodes to be averaged.

In an effort to impose further clustering on the database, the robot control software now attempts to find similar nodes within the database and endow the current node with characteristics similar to these nodes. It does so with the help of the comparison operators defined above, which are used to find a specified number of nearest matches to the current node. Relations from these similar nodes are then explicitly added to the current node. Care must be taken here, as this might result in multiple insertions of the same relation into the same node. To understand how this is accounted for, it is necessary to digress briefly into the more practical aspects of this scheme.

The method described thus far is impractical. Without limits on the number of relations a node can contain, they will become larger and larger with each training iteration—so large as to render practical use of the scheme impractical in a very short time. Thus, the nodes used in practice limit themselves to containing a specified number of the strongest relations. Inserting a new relation into this node works only if the relation is stronger than the weakest relation already present. The only exceptions to this rule are the sensor and activator nodes. The latter at least must remain present for future actions to be derived from information stored in the graph. In any case, when two like relations are inserted into

a node, the one with the highest relatedness value wins.

Regardless of its advantages, this limiting strategy calls some of the founding principles of this research into question. In the modern associational representational schema described thus far, a concept is represented by dint of patterns formed in its representation's relations—strong ones as well as weak ones. Does the decision to exclude weak relations compromise the representational ability of massively interlinked structures? The experiments described here offer no definitive answers. However, their moderate success suggests that the representation used by these structures is of a slightly different, albeit still useful type. Instead of consisting of a large set of greatly varying relations, these relations are perhaps better thought of as nodes on an unweighted graph. Their meaning is encoded in their limited and particular connectivity, rather than a pattern of weights. While this may not be as effective an encoding overall, its greater speed and smaller size is surely an asset for real-time applications.

The mechanism for adding new relations to the current node is nearly fully explained. There is only one problem remaining—namely that all the schema described above for adding new relations to the current node must extract them from already-existing nodes. This means that the nodes created first would only have actuator and sensor relations, and likewise with nodes created after those nodes, and so on. There needs to be a way to stimulate the immediate creation of new relations. The method used at present simply compares the current node to the rest of the nodes in the graph and adds a relation to its nearest match. Once this node is inserted into the database, this relation becomes part of future current nodes if the node is ever used in their creation.

As a final note, it is worth mentioning that all or none of these stages may be omitted from the training and execution phases at the discretion of the operator. The method of specifying which to include is described in the next section.

3.5 Hardware and software description

The hardware used throughout the following experiments was a Sun Ultra 10 workstation with a 440 megahertz UltraSPARC IIi processor. This stationary machine controlled an ActivMedia AmigoBOT robot over a 9,600 bps radio modem. The AmigoBOT is a simple two-wheeled robot with four forward-looking sonars, two rear-looking sonars, and two side-looking sonars. This platform is familiar to anyone who has used ActivMedia's Pioneer line of robots.

The software used to control the AmigoBOT was a custom C++ program built atop the Saphira system, once the

primary control software for most ActivMedia robots.[7] Saphira was used entirely for its convenient interface to the robot's sensors and motors—the rest of its functionality, including mapping and obstacle avoidance behaviors, was disabled.

The essential functioning of the robot control software proceeded as follows:

1. Initialize Saphira
2. Set up internal objects and data structures
3. If in the training phase, repeat:
   (a) Execute keyboard teleoperation commands
   (b) Collect robot inputs and outputs from Saphira
   (c) Create a new graph node (the current node) based on the present robot state and existing experience in the graph
   (d) If appropriate, keep the new node in the graph
   (e) Monitor for the command to switch to execution phase

   If in the execution phase, repeat:
   (a) Collect robot inputs from Saphira
   (b) Create a new graph node (the current node) based on sensor inputs and existing experience in the graph
   (c) Derive and execute motor outputs for the new current node
   (d) If appropriate, keep the new node in the graph
   (e) Monitor for the command to switch to training phase

The final two loops describe the training and execution phases. In the current version of the control software, these repeat at no specific interval but rather as fast as they can. The runtimes of the loops scale linearly with the size of the database, as each glance at prior experience invokes the same number of calls to the various routines that search the entire graph for the nearest matches to the current node. It is understood that this simplified approach might compromise the learning ability of the scheme, since concepts learned when the database was leaner might rely on a faster sampling rate than the program is capable of providing later on. The last section of this paper contains some suggestions for possible performance improvements that might help alleviate this problem.

[7] Saphira has largely been superseded by a software package named ARIA, which was not available for Sun systems at the time of research.

3.6 Experiments with stateless tasks

With all this infrastructure defined, the purpose of the following experiments is to demonstrate that it is better than straightforward instance-based learning at acquiring certain stateless tasks. At present, it would be reckless to state that this is absolutely the case. However, the experiments do appear to show that a robot trained under the scheme presented above has a slightly greater degree of situational awareness than a robot trained with instance-based learning. This is to say that in some of the tasks that require the robot to behave differently based on certain immediate cues (e.g. the nature of the environment around the robot), the robot trained under the above scheme appears to reproduce the task more faithfully than robots trained under straightforward instance-based learning. This result is not entirely conclusive, however, and needs further investigation before it can be presented as fact.

All of the experiments used the same basic training and execution phase loops. First, within the execution phase, the control program never stored new information in the graph (the chief means of keeping state in stateful tasks). Second, the control program never used the immediate historical context in creating the nodes, mostly to establish whether the "enforced clustering" technique of borrowing relations from similar nodes was helpful on its own. Lastly, when training the robot with straightforward instance-based learning for comparison's sake, the lists were pared down solely to storing sensor and actuator values in the training phase and, in the execution phase, storing sensor values and deriving actuator values.

All experiments took place in the Swarthmore College Computer Science Department's Sun Lab in Sproul Hall. The area allocated for the robot was roughly in the shape of an irregular pentagon, with three perpendicular sides and two sides forming a low "peak" opposite the longest of these. The perpendicular sides described an area approximately ten feet long and five feet wide, while the width at the peak increased to around eight feet. The edges of the area were delineated by walls and foamcore board on the perpendicular sides, while the remaining two sides were delineated by an upturned small table and dense masses of stout table and chair legs. A rough plan of this environment appears in Figure 14.

3.6.1 Test of functionality: Obstacle avoidance

The first experiment was not a comparative experiment; it was a test of the system's basic functionality. The robot was trained to move continually without hitting obstacles. In this experiment, nodes were permitted to contain a maximum of 30 relations. In creating new nodes, the control program drew relations from five similar nodes from the database. When deriving outputs in the execution phase, the control program averaged the outputs of five similar nodes.

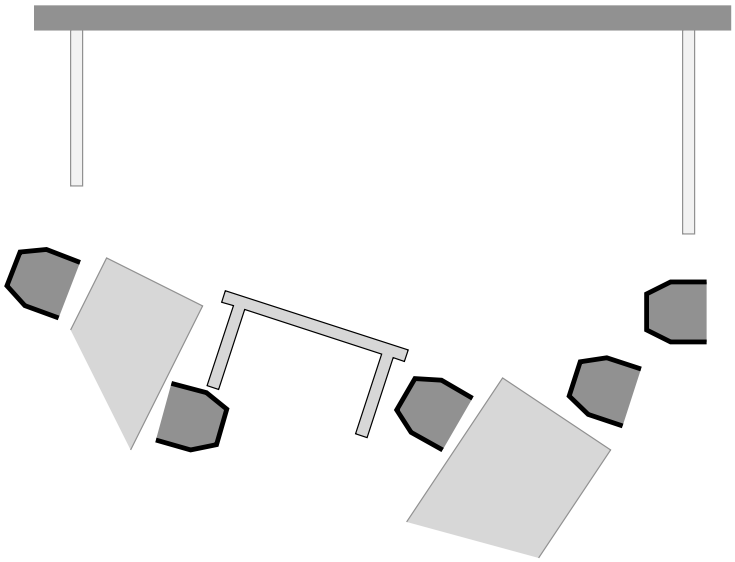

Figure 14: A rough floor plan of the robot environment. Chairs, a wall, table legs, foamcore panels, and an upturned table delineated the space.

The robot was trained for approximately 480 iterations of the training phase loop (around a minute and a half). As the number of iterations increased, each training phase loop iteration became progressively slower, and at the end of training they took somewhere between a half-second and a second to complete. Training was conducted by manually driving the robot around the pentagonal area and, when it drew near to an edge, briefly slowing it down and turning it to the right.

Once in execution mode, the robot performed the task adequately. It tended to proceed forward with a slight rightward turn when in open space, moving at roughly the same speed as it did during training. When it approached an obstacle, it slowed down and turned to the right. Nearer obstacles provoked more acute turns on several occasions. Unfortunately, this was not the case on every occasion, and the test was concluded when the robot grazed the wall with its right fender.

The results of this experiment were deemed satisfactory enough to allow work to continue onward with more difficult tasks. For these, as mentioned above, the goal is not necessary to establish whether the learning scheme works itself, but whether it offers any improvement over straightforward instance-based learning.

3.6.2 First task: Perimeter following

For this task, the robot was trained to follow the perimeter of the pentagonal work area at a distance ranging from one to two feet.

The straightforward instance-based learning control was trained first. It was found here that this sort of training without any limits on its speed quickly fills the graph with thousands of nodes, so both training and execution phase loops were supplemented with a half-second pause in program execution. When deriving outputs in the execution phase, the control program averaged the outputs of five similar nodes.

The robot was driven for five loops around the perimeter of the pentagonal work area. Despite the delay, the training phase loop completed on the order of 700 iterations. Once switched to execution mode, the robot began to pursue a general and increasingly tight curving path, paying less and less attention to the features of the perimeter. It did continue to describe a looping path, however, and avoided hitting any obstacles throughout the test, which consisted of approximately five circuits.

The learning scheme described above was attempted next. Here the control program was made to derive new relations from five similar nodes and derive outputs from five similar nodes in the execution phase. The robot was driven for five loops around the pentagonal work area, completing around 300 iterations of the training phase loop. When placed in execution mode, the robot struck a wall almost immediately. The robot was trained again, this time completing enough loops for over 500 iterations of the training phase loop. Once in execution mode, it remained near the perimeter for two circuits, proceeding without turning along the longer straight sides of the area. Its turns were longer and less acute than what it experienced in training. Eventually it drew further away from the wall and began to describe a more general, circular path.

The performance of the newer learning scheme has been repeated, but not with absolute consistency. However, when it did work, it seemed to demonstrate an organization of the task that the straightforward instance-based learning based robot lacked. This is more notable in light of the differences in the perimeter delineation of the work area, which was solid wall on only some sides and intermittent table and chair legs on others.

3.6.3 Second task: Obstacle orbiting

This more difficult task requires the robot to circle two obstacles within the work area. The obstacles, a 2.5 foot wide circular trash bin and a stack of 8.5” by 7” programming references, were placed a foot apart in the middle of the area, oriented parallel to its longest side. There was at least two feet of clearance on all sides of the obstacle.

A small variety of different configurations were tried for both straightforward instance-based learning and the newer learning scheme, as none of them worked successfully. In each case, the training was identical: the robot was hand driven around the obstacles five times and then placed into execution mode. In both cases, the most successful run saw the robot circle around the trash bin and continue circling into the books. Typically runs trained with straightforward instance-based learning saw the robot undershoot the turn around the trash bin and run into it, while runs trained with the newer scheme generally saw the robot overshoot the turn and head off into a corner.

For the best run made by a straightforward instance-based learning trained robot, the control program was made to derive new outputs from five similar nodes. For the best run made by a robot trained under the newer scheme, the control program was made to derive new relations from five similar nodes when creating new nodes, and to derive new outputs from five similar nodes.

3.6.4 Third task: Figure eights around obstacles

One final task was tried under both learning methods. The obstacles from the last experiment were spread out to a distance of 2.5 feet while maintaining the two feet of clearance on its sides. Here, the robot had to execute a figure eight around the obstacles, making a left turn around the book stack and a right turn around the trash bin. It is arguable that this task is pushing the limits of what could be accomplished without any state—certainly it imposes a hefty burden on the control system of discriminating between fine differences in sensor input, and there is a chance that aspects of both input and control are too imprecise for the task without the aid of state information. Nonetheless, it was attempted.

Again, a variety of different configurations were tried, including the familiar five similar nodes for new relations and five similar nodes for new outputs. Both straightforward instance-based learning training sessions and newer method training sessions involved the robot being led through five figure eights. Unfortunately, all runs failed in a rather consistent fashion. Put into execution mode as it was directly in between the obstacles, close to the point to begin a left turn around the book stack, the robot continued on its own by moving directly forward, beginning to turn left only when it came within about a foot the perimeter of the work area.

3.7 Experiments with stateful tasks

The learning method described above has shown lackluster performance in learning stateless tasks. Nevertheless, it is valuable to find out whether it can learn information of a fundamentally different type: state. If there is no constant

sensor cue that informs a robot which part of a task it must be performing at any time, then it must maintain this information within itself. This is in contrast with the last two tasks above—even though they consisted of distinct stages (e.g. times of turning and going straight in the obstacle orbiting task, or times of turning right and turning left in the figure eight task), there were, at least in theory, environmental cues from the obstacles that let the robot know exactly which stage it should be pursuing. (Again, this may not have been entirely the case for the third task.) That the robot failed to acquire these tasks without provisions for acquiring state information might be significant.

The two tasks described below were designed to be free of environmental cues. However, it is not possible to totally separate environmental cues from the operation of the robot, so the (very marginal) success of the robot at these tasks might not be entirely due to statekeeping. As such, in light of this and the limited performance demonstrated by the system, it would be unwise here as well to state unequivocally that the robot is successfully keeping track of task state. Still, there are signs that it may have an inkling of what it should be doing.

Before the experiments can be described, the present method for statekeeping under the newer learning scheme should be described. It involves the full node construction and insertion scheme described in the fourth section of this paper, where relations from recently created nodes are also inserted into the current node and then the current node itself is (potentially) inserted into the graph. What's important is why all this is done. The relations from recently created nodes are added to encourage large scale clustering of relatively contemporaneous nodes, hopefully facilitating the creation of state clusters within the database. New nodes are then inserted into the database mostly due to memory management concerns. At present, if a node is not inserted into the database, it is deleted. Because it's not possible to extract the relations of recently created nodes if the nodes have been destroyed, all the nodes are currently saved.

Unlike the stateless task experiments, the following experiments do not feature a control based on a similar, familiar learning method. Put simply, there isn't one. Because straightforward instance-based learning requires immediate sensory cues to determine what action to perform, and because no sensory cue can indicate the current state of the task, this method simply can't do the task. Instead, the goal of these experiments is simply to coax the newer learning method into working at all.

3.7.1 Figure eights in an open area

The first task attempted required the robot to make figure eights in the open work area described earlier. This time, some of the barriers were not present. In training, the robot was led between five and ten through a figure eight around seven feet long and between three and four feet wide.

The robot was switched into the execution phase at the bottom of the figure eight. It continued to turn, then correctly proceeded straight for a few feet, then began turning in the other direction. Unfortunately it persisted in looping in this direction until the experiment was brought to a close.

This behavior was consistent across several different configurations of the control program.

3.7.2 Alternating rotation

With figure eights bringing continuing bad luck, the next experiment was designed to present the robot with a somewhat easier task. Here, the robot was placed in front of the wall and trained to make a revolution in one direction, then a revolution in the other, over and over. The wall was intended to relieve the robot of having to measure out exactly 360 degrees on each rotation, though it could not offer any hints about which direction the robot had to turn at any given moment.

In training, the robot was led through approximately ten rotation pairs. When the last rotation pair was over, and the robot was not moving, the control program was switched to the execution phase. The robot subsequently executed one full revolution in the appropriate direction, then stopped and began to revolve in the other. Unfortunately, it continued to revolve in this direction until the experiment was terminated.

Again, this behavior was consistent across several different configurations of the control program.

3.8 Robot control conclusions

The experiments above may have demonstrated a glimmer of functionality. However, they also demonstrate that the present system has some significant practical and methodological flaws.

The practical flaws are easy to spot, though it's much harder to determine exactly how they affect the system as a whole. Perhaps the largest one is the increasing slowdown involved in creating nodes by searching for similar representations within the database. The sample rate gets rather slow in a very short time, diminishing to somewhere around 1 Hz. This low frequency could be quite detrimental to the operation of the system, as the AmigoBOT operating at regular training speeds can move over half a foot in that time. This aside, the varying times between samples could also compromise the learning ability of the system.

Other practical flaws concern the implementation of the methods described above. At present, objects representing nodes are identified globally by their memory addresses. This makes removing nodes from the graph a bad idea, since a relation to a removed node could potentially refer in time to another node that takes the memory space of the old

one. Similar little problems like these are not uncommon throughout the system.

Potential remedies to these problems include the following:

1. To find a number of nodes similar to a particular node, the present program simply compares the node to all the nodes in the database. It would be much faster to use the *k*d-tree technique described above to locate similar nodes. This technique, combined with more efficient programming and perhaps faster hardware, might improve search speed significantly. Ideally, search time would be reduced to the point where the program can insert delays in order to maintain a fast, consistent sample rate.
2. Naturally, better programming would help with the second flaw.

The methodological flaws in the method above are more difficult to spot. A sampling of the larger ones is presented here with the tacit warning that they may only be the beginning.

First, there are certain analogies that the system can't make. Hypothetically, if the robot were trained on a simple obstacle avoidance task, and one sonar rather miraculously does not note any near obstacles at all throughout training. Later, during the execution phase, this sonar begins functioning correctly and is the only one to take notice of a pole in the robot's path. Ideally, the robot should have figured out that any obstacle is a cause for turning. In most actual training instances, it would by dint of having something appear in front of the sonars at some point or another. However, there's really no way for the system to make those sorts of analogies. This might be a rather serious deficiency.

Second, it's difficult or impossible under the current scheme to teach the robot what it should not do. When demonstrating a task, often the operator doesn't have the opportunity to show how the robot should handle abnormal situations, such as a person walking into the work area, or, more mundanely, such as what to do when motor skew puts the robot somewhere it's never been before.

Third, all of the relations within a node have equal importance. A particular sensor input could signify the need for an important change in the behavior of the robot—for instance, the example again where a single sensor detects a very thin object in the path of the robot. Yet the relation corresponding to that sensor may be one of hundreds in the current node. How can its importance be emphasized?

Fourth, state information storage under the present system is somewhat balky, as it requires new information to be continually stored in the database (or at least preserved temporarily) in order to store information about the robot's temporal context in new nodes. This is more efficient, both in memory usage and computation time, if nodes that are deemed to similar to ones already stored can die out. Unfortunately, as mentioned previously, the current system does not permit deletion of nodes. This, compounded with the already lackluster performance of the system on stateful tasks, makes the current mode of storing state information seem somewhat questionable.

It's difficult to see straightaway how these problems can be solved, though some seem more approachable than others. They need not spell disaster for using massively interlinked representations in robot control. However, they should be seriously considered by future endeavors with like techniques.

4 Conclusion

We have seen that associational computing—the use of massively interlinked graph structures in the creation, manipulation, and application of concept representations—can be used with some success in two very different domains. It seems clear, however, that this approach is more easily applied to situations in which the concepts to be represented can be easily divided into discrete units. Trying to divine discrete structure from more complex continuous information is a difficult task in itself, and one whose success here has been rather limited.

In any case, it is hoped that the methods described here have shown some glimmer of utility. It could be that their most effective use comes in tandem with other information representation technologies—that they can supplement other methods in order to form a more complete mechanism for representing complex concepts.

References

[1] David Hume, *A Treatise of Human Nature, vol. 1*. L. A. Selby Bigge, ed. New York: E. P. Dutton & Co. Inc. (1949)

[2] Noam Chomsky, "A review of B.F. Skinner's *Verbal Behavior*." *Language* 35 (1959): 26-58.

[3] Douglas B. Lenat, Ramanathan V. Guha, Karen Pittman, Dexter Pratt, and Mary Shepherd, "Cyc: Toward Programs with Common Sense." *Communications of the ACM*, 33(8) (1990): 30-49.

[4] David S. Touretzky and Lisa M. Saksida, "Operant Conditioning in Skinnerbots." *Adaptive Behavior*, 5(3/4) (1997): 219-247.

[5] G.A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K.J. Miller, "Introduction to WordNet: An On-line Lexical Database", *Journal of Lexicography*, 5(4) (1990): 234-244.

[6] Kevin Knight, "Learning Word Meanings by Instruction" *AAAI/IAAI*, 1 (1996): 447-454.

[7] Cynthia A. Thompson, "Acquisition of a Lexicon from Semantic Representations of Sentences" *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, (1995): 335-337.

[8] Jeffrey L. Elman "Finding Structure in Time", *Cognitive Science*, 14(2) (1990): 179-211.

[9] Douglas S. Blank, Lisa A. Meeden, and James Marshall, "Exploring the Symbolic/Subsymbolic Continuum: A case study of RAAM" *The Symbolic and Connectionist Paradigms: Closing the Gap*, (1992): 113-148

[10] Risto Miikkulainen and Michael G. Dyer, "Natural Language Processing With Modular PDP Networks and Distributed Lexicon" *Cognitive Science*, 15(3) (1991): 343-399.

[11] Tony Plate, "Estimating analogical similarity by vector dot-products of Holographic Reduced Representations" Unpublished manuscript (1995).

[12] Paul Smolensky, "Tensor product variable binding and the representation of symbolic structures in connectionist networks" *Artificial Intelligence*, 46 (1990): 159-216

[13] Burghard B. Rieger, "Fuzzy word meaning analysis and representation in linguistic semantics: an empirical approach to the reconstruction of lexical meanings in East- and West-German newspaper texts" *COLING-80*, (1980): 76-84.

[14] Burghard B. Rieger, "Inducing a Relevance Relation in a Distance-like Data Structure of Fuzzy Word Meaning Representations", *4th International Conference on Data Bases in the Humanities and Social Sciences.*, (1985): 374-386.

[15] Kurt Burgess and Kevin Lund, "Producing high-dimensional semantic spaces from lexical co-occurrence", *Behavior Research Methods, Instruments, & Computers*, 28 (1996): 203-208.

[16] Thomas K. Landauer, Peter W. Foltz, and Darrell Laham, "Introduction to Latent Semantic Analysis". *Discourse Processes*, 25 (1998) 259-284.

[17] Andrew W. Moore and Kan Deng, "Multiresolution Instance-based Learning" *Proceedings of the International Joint Conference on Artificial Intelligence*, (1995)

[18] Lisa A. Meeden, G. McGraw, and Douglas S. Blank, "Emergent control and planning in an autonomous vehicle". *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, (1993) 735-740.

[19] Tim Oates "Grounding Knowledge In Sensors: Unsupervised Learning for Language and Planning". Ph.D. dissertation, University of Massachusetts, Amherst. (2001)

Appendix A: Code

A.1 Summer 2000 research code

MisterScan2.c

This file created the raw ASCII words database files by scanning a file containing the concatenated contents of the Project Gutenberg corpus. It was executed once for each word.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <search.h>

#define SHMUTZ 0
#define DATA 1
/* viva Perl */
#define UNLESS(item)   if(!( (item) ))

char *progname = "MisterScan";
char *theword;

/**********
 ** Fatalism.
 **********/

void die(char *why)
{
  fprintf(stderr, "%s: %s.\n", progname, why);
  exit(-1);
}

void mem_die(char *why)
{
  fprintf(stderr, "%s: couldn't allocate memory for %s.\n", progname, why);
  exit(-1);
}

void file_die(char *why)
{
  fprintf(stderr, "%s: couldn't open %s.\n", progname, why);
  exit(-1);
}

/**********
 ** Queue.
 **********/

typedef struct _Queue_elem {
  struct _Queue_elem *next;
  char *datum;
} Queue_elem;

typedef struct _Queue {
  Queue_elem *begin, *end;
} *Queue;

Queue Queue_new(void)
{
  Queue q;

  UNLESS(q = malloc(sizeof(struct _Queue))) mem_die("new queue");
  q->begin = q->end = NULL;

  return(q);
}
```

```
void Queue_enqueue(Queue q, char *datum)
{
  Queue_elem *qe;

  UNLESS(qe = malloc(sizeof(Queue_elem))) mem_die("new queue element");
  if(q->end) q->end->next = qe;
    else q->begin = qe;
  q->end = qe;

  qe->next = NULL;
  qe->datum = datum;
}

char *Queue_dequeue(Queue q)
{
  char *result;
  Queue_elem *old;

  UNLESS(old = q->begin) return(NULL);
  result = old->datum;
  if(q->begin == q->end) q->begin = q->end = NULL;
    else q->begin = old->next;

  free(old);
  return(result);
}

/**********
 ** Parameters.
 **********/
struct Params {
  char *words;
  Queue corpi;
  long bufsiz;
  int qlength;
  char *outdr;
} cf;

void getParams(char *fn)
{
  FILE *pf;
  char *corpus;

  cf.corpi = Queue_new();

  UNLESS(pf = fopen(fn, "r")) file_die("the config file");

  UNLESS(fscanf(pf, "%as %as %ld %d", &cf.words,
           &cf.outdr, &cf.bufsiz, &cf.qlength) == 4) die("bad config file");

  while(fscanf(pf, "%as", &corpus) == 1) Queue_enqueue(cf.corpi, corpus);

  fclose(pf);
}

/**********
 ** Words loader.
 **********/

char **words = NULL;
unsigned int wordcount = 1;

void readWords(void)
{
  FILE *wf;
  ENTRY e;
  char *wd, *wdp, **wdsp;

  UNLESS(wf = fopen(cf.words, "r")) file_die("the words database");

  while(fscanf(wf, "%as", &wd) == 1) {
    wdp = wd;
```

```
    while(*wdp) *wdp = tolower(*wdp++);

    UNLESS(words = realloc(words, ++wordcount * sizeof(char *)))
      mem_die("word list");
    words[wordcount - 2] = wd;
  }

  words[wordcount - 1] = NULL;
  fclose(wf);

  UNLESS(hcreate(wordcount)) mem_die("words hashtable");
  wdsp = words;
  while((e.key = *wdsp++)) {
    UNLESS(e.data = (char *) malloc(sizeof(unsigned long int)))
      die("word hash table value");
    *((unsigned long int *) e.data) = 0;
    UNLESS(hsearch(e, ENTER)) mem_die("hash entry");
  }

  /* Check to see if the word is actually in the database */
  e.key = theword;
  UNLESS(hsearch(e, FIND)) die("Word is not in words database");
}

/**********
 ** Corpus loader.
 **********/

char *BigBufr;
char **WdPtrs = NULL;

void initBigBufr(void)
{
  UNLESS(BigBufr = malloc(sizeof(char) * cf.bufsiz)) mem_die("book buffer");
}

void loadBigBufr(FILE *fp)
{
  char c, *cp;
  int readmode = SHMUTZ, ptrct = 1;
  long int i;

  cp = BigBufr;
  if(WdPtrs) free(WdPtrs);
  UNLESS(WdPtrs = malloc(sizeof(char *))) mem_die("pathetic one (char *)");

  for(i=0; i<cf.bufsiz-1; i++) {
    if((c = fgetc(fp)) == EOF) {
      WdPtrs[ptrct - 1] = NULL;
      *cp++ = '\0';
      break;
    }
    else {
      if(readmode == SHMUTZ) {
        if(isalnum(c)) {
          readmode = DATA;
          *cp = tolower(c);

          WdPtrs[ptrct++ - 1] = cp++;
          UNLESS(WdPtrs = realloc(WdPtrs, ptrct * sizeof(char *)))
            mem_die("word pointer array");
        }
        continue;
      }
      /* readmode must equal DATA now */
      if(isalnum(c)) *cp++ = tolower(c);
      else {
        *cp++ = '\0';
        readmode = SHMUTZ;
      }
    }
  }
```

```
  *cp = '\0';
}

/**********
 ** Corpi scanner
 **********/

void scanBuffer(void)
{
  ENTRY e, *ep;
  char **f_wd, **m_wd, **l_wd;     /* front, mid, and last words */
  char **cp;
  int i;

  f_wd = m_wd = l_wd = WdPtrs;
  for(i=0; i<cf.qlength; i++) if(*f_wd) f_wd++;
  for(i=0; i<(cf.qlength/2); i++) if(*m_wd) m_wd++;

  while(*f_wd) {
    if(!strcmp(*m_wd, theword)) {
      cp = l_wd;

      while(cp != f_wd) {
        e.key = *cp;
        if((ep = hsearch(e, FIND))) *((unsigned long int *) ep->data) += 1;
        cp++;
      }
    }

    f_wd++; m_wd++; l_wd++;
  }
}

void scanCorpi(void)
{
  FILE *corpus;
  char *fn;

  initBigBufr();

  while((fn = Queue_dequeue(cf.corpi))) {
    UNLESS(corpus = fopen(fn, "r")) file_die(fn);

    while(!feof(corpus)) {
      loadBigBufr(corpus);
      scanBuffer();
    }
  }
}

/**********
 ** Data saver
 **********/

void writeData(void)
{
  FILE *fp;
  char **wp;
  unsigned long int *num, *div;
  ENTRY e, *ep;

  chdir(cf.outdr);
  UNLESS(fp = fopen(theword, "w")) file_die("database output file");

  e.key = theword;
  UNLESS(ep = hsearch(e, FIND)) die("Word is not in words database");
  div = (unsigned long int *) ep->data;

  wp = words;
  while((e.key = *wp++)) {
    UNLESS(ep = hsearch(e, FIND)) die("Word should be in hash, but isn't");
```

```
    num = (unsigned long int *) ep->data;
    fprintf(fp, "%.10f %s\n",
                    *div == 0 ? (double) 0 : (double) (*num)/(*div), e.key);
  }

  fclose(fp);
}


/**********
 ** Number One
 **********/

int main(int argc, char **argv)
{
  if(argc != 3) die("usage: MisterScan conffile word");

  theword = argv[2];

  getParams(argv[1]);
  readWords();
  scanCorpi();
  writeData();

  return(0);
}
```

NewDB.c

This file contains the library code responsible for manipulating and performing computations on the binary representation words database. It operated at the heart of all programs made during the summer of 2000. As such, its general sprawl reflects the twists and turns of research over the course of those months.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <math.h>

#include <gdbm.h>

/* Important database filenames.
 *
 * Change the size of the constant MAX_FN_LENGTH below if
 * any of these exceed 13 characters in length. */
#define DB_REVERSE_INDEX_FILE "REVINDEX"
#define DB_WORDLIST_FILE      "WORDLIST"
#define DB_POPULARITY_FILE    "POPULARITY"

/* Miscellaneous constants */
#define MAX_WD_LENGTH         40
#define MAX_FN_LENGTH         15

/* Perl does a few things right. */
#define UNLESS(item)   if(!( (item) ))

typedef double *DB_CompArray;

typedef struct _DB_RawArray_rec {
  char *word;
  unsigned long int *array;
} DB_RawArray_rec;

typedef DB_RawArray_rec *DB_RawArray;

typedef struct _DB_Info_rec {
  char *dbpath;
  GDBM_FILE popularity;
  char **wordlist;
  unsigned long int numwords;
} DB_Info_rec;

typedef DB_Info_rec *DB_Info;

/* Miscellaneous functions */
#ifdef NEWDB_PERL_MODULE
char *progname = "Perl program";
#else
extern char *progname;
#endif

void die(char *why)
{
  fprintf(stderr, "%s: %s.\n", progname, why);
  exit(-1);
}

void mem_die(char *why)
{
  fprintf(stderr, "%s: unable to allocate memory for %s.\n", progname, why);
  exit(-1);
}

void file_die(char *why)
{
  fprintf(stderr, "%s: cannot open file %s.\n", progname, why);
  exit(-1);
}
```

```
char *get_stringspace(int length)
{
  char *buf;

  UNLESS(buf = malloc(length * sizeof(char))) mem_die("a string");
  return buf;
}

/* A necessary function declaration */
DB_RawArray NewDBLoadArray(DB_Info info, char *word);

/* And some others, for eliminating arrays */
void NewDBZapRawArray(DB_RawArray ra)
{
  free(ra->word);
  free(ra->array);
  free(ra);
}

void NewDBZapCompArray(DB_CompArray ca)
{
  free(ca);
}

/* DB init routine */

DB_Info NewDBinit(char *DBpath)
{
  DB_Info info;
  char *temp;
  int pathlen;
  FILE *wdfile;
  unsigned long int wdcount=0;

  UNLESS(info = malloc(sizeof(struct _DB_Info_rec)))
    mem_die("DB_Info record");
  pathlen = strlen(DBpath) + MAX_FN_LENGTH;

  /* Now we start to fill out the info record. */
  info->dbpath = get_stringspace(strlen(DBpath) + 1);
  strcpy(info->dbpath, DBpath);

  /* Open the word popularity database file */
  temp = get_stringspace(pathlen);
  sprintf(temp, "%s/%s", DBpath, DB_POPULARITY_FILE);
  UNLESS(info->popularity = gdbm_open(temp, 0, GDBM_READER, S_IRUSR, 0))
    file_die(temp);

  /* Finally, load the list of words. */
  sprintf(temp, "%s/%s", DBpath, DB_WORDLIST_FILE);
  UNLESS(wdfile = fopen(temp, "r")) file_die(temp);

  free(temp);
  info->wordlist = NULL;

  for(;;) {
    if(feof(wdfile)) die("DB_WORDLIST_FILE is improperly formatted");

    temp = get_stringspace(MAX_WD_LENGTH);
    if(fscanf(wdfile, "%sMAX_WD_LENGTH", temp) == EOF) {
      free(temp);
      break;
    } else {
      UNLESS(info->wordlist = realloc(info->wordlist,
                ++wdcount*sizeof(char *)))
        mem_die("the list of words");
      info->wordlist[wdcount - 1] = temp;
    }
  }

  fclose(wdfile);
  info->numwords = wdcount;
```

```
  return(info);
}

/* Close the database */
/* close all opened DB files, purge information in the
 * DB_Info data structure */
void NewDBclose(DB_Info info)
{
  unsigned long int i;

  free(info->dbpath);
  gdbm_close(info->popularity);

  for(i=0; i<info->numwords; i++) free(info->wordlist[i]);
  free(info->wordlist);

  free(info);
}

/* The word usage array loader */
DB_RawArray NewDBLoadArray(DB_Info info, char *word)
{
  char *temp;
  DB_RawArray newarray;
  unsigned long int wdcount=0;
  FILE *arrayFile;

  temp = get_stringspace(strlen(info->dbpath)+strlen(word)+2);
  sprintf(temp, "%s/%s", info->dbpath, word);
  arrayFile = fopen(temp, "r");
  free(temp);
  UNLESS(arrayFile) {
    fprintf(stderr, "%s: %s: wordfile does not exist in %s\n",
    progname, word, info->dbpath);
    return(NULL);
  }

  UNLESS(newarray = malloc(sizeof(struct _DB_RawArray_rec)))
    mem_die("a new DB_RawArray record");

  newarray->word = get_stringspace(strlen(word)+1);
  strcpy(newarray->word, word);

  UNLESS(newarray->array = malloc(info->numwords * sizeof(unsigned long int)))
    mem_die("a new raw array's data space");
  wdcount = fread(newarray->array,
  sizeof(unsigned long int),
  info->numwords,
  arrayFile);

  fclose(arrayFile);


  if(wdcount != info->numwords) {
    fprintf(stderr, "NewDBlib error: The number of array entries in:\n");
    fprintf(stderr, "  %s\n", word);
    fprintf(stderr,
            "is %ld, which does not match the number of words (%ld)!\n",
    wdcount, info->numwords);
  }

  return(newarray);
}

DB_CompArray NewDBAllocScalarArray(DB_Info info, double value)
{
  DB_CompArray newarray;
  unsigned long int i;

  UNLESS(newarray = malloc(info->numwords * sizeof(double)))
    mem_die("a new scalar comparray");
```

```
  for(i=0; i<info->numwords; i++) newarray[i] = value;

  return(newarray);
}

DB_CompArray NewDBAllocZeroArray(DB_Info info)
{
  return(NewDBAllocScalarArray(info, 0));
}

DB_CompArray NewDBComputeArray(DB_Info info, DB_RawArray rawarray)
{
  datum key, val;
  double *newarray, *na_runner;
  unsigned long int *ra_runner, popularity, i;

  UNLESS(newarray = malloc(info->numwords * sizeof(double)))
    mem_die("a new computed array");

  key.dptr = rawarray->word;
  key.dsize = strlen(rawarray->word)*sizeof(char);

  val = gdbm_fetch(info->popularity, key);
  if(val.dptr == NULL) {
    fprintf(stderr, "\n\n%s\n", key.dptr);
    die("can't find word in popularity hash!");
  }
  popularity = *((long*) val.dptr);
  free(val.dptr);

  na_runner = newarray;
  ra_runner = rawarray->array;

  if(popularity == 0) for(i=0; i<info->numwords; i++) *na_runner++ = 0;
    else for(i=0; i<info->numwords; i++)
           *na_runner++ = (double) *ra_runner++ / (double) popularity;

  return(newarray);
}

double NewDBIntegrateArray(DB_Info info, DB_CompArray comparray)
{
  double *ca_runner, result=0;
  unsigned long int i;

  ca_runner = comparray;
  for(i=0; i<info->numwords; i++) result += *ca_runner++;

  return(result);
}

DB_CompArray NewDBCompArrayDup(DB_Info info, DB_CompArray comparray)
{
  DB_CompArray newarray, na_runner, ca_runner;
  unsigned long int i;

  UNLESS(newarray = malloc(info->numwords * sizeof(double)))
    mem_die("a computed array");

  na_runner = newarray;
  ca_runner = comparray;

  for(i=0; i<info->numwords; i++) *na_runner++ = *ca_runner++;

  return(newarray);
}

DB_CompArray NewDBThreshold_(DB_Info info, DB_CompArray comparray,
     double threshold)
{
  double *ca_runner;
  unsigned long int i;
```

```
  ca_runner = comparray;
  for(i=0; i<info->numwords; i++) {
    if(*ca_runner < threshold) *ca_runner = 0;
    ca_runner++;
  }

  return(comparray);
}

DB_CompArray NewDBThreshold(DB_Info info, DB_CompArray comparray,
                            double threshold)
{
  return NewDBThreshold_(info, NewDBCompArrayDup(info, comparray), threshold);
}

DB_CompArray NewDBNegativeArray_(DB_Info info, DB_CompArray comparray)
{
  double *ca_runner;
  unsigned long int i;

  ca_runner = comparray;
  for(i=0; i<info->numwords; i++) *ca_runner++ = -*ca_runner;
  return(comparray);
}

DB_CompArray NewDBNegativeArray(DB_Info info, DB_CompArray comparray)
{
  return NewDBNegativeArray_(info, NewDBCompArrayDup(info, comparray));
}

DB_CompArray NewDBAbsArray_(DB_Info info, DB_CompArray comparray)
{
  double *ca_runner;
  unsigned long int i;

  ca_runner = comparray;
  for(i=0; i<info->numwords; i++) {
    if(*ca_runner < 0) *ca_runner = -*ca_runner;
    ca_runner++;
  }
  return(comparray);
}

DB_CompArray NewDBAbsArray(DB_Info info, DB_CompArray comparray)
{
  return NewDBAbsArray_(info, NewDBCompArrayDup(info, comparray));
}

DB_CompArray NewDBSumArray_(DB_Info info, DB_CompArray comparray1,
  DB_CompArray comparray2)
{
  double *ca1_runner, *ca2_runner;
  unsigned long int i;

  ca1_runner = comparray1;
  ca2_runner = comparray2;

  for(i=0; i<info->numwords; i++) *ca1_runner++ += *ca2_runner++;

  return(comparray1);
}

DB_CompArray NewDBSumArray(DB_Info info, DB_CompArray comparray1,
                                         DB_CompArray comparray2)
{
  return NewDBSumArray_(info,
NewDBCompArrayDup(info, comparray1),
comparray2);
}

DB_CompArray NewDBMultArray_(DB_Info info, DB_CompArray comparray1,
```

```
   DB_CompArray comparray2)
{
  double *ca1_runner, *ca2_runner;
  unsigned long int i;

  ca1_runner = comparray1;
  ca2_runner = comparray2;

  for(i=0; i<info->numwords; i++) *ca1_runner++ *= *ca2_runner++;

  return(comparray1);
}

DB_CompArray NewDBMultArray(DB_Info info, DB_CompArray comparray1,
                                          DB_CompArray comparray2)
{
  return NewDBMultArray_(info,
   NewDBCompArrayDup(info, comparray1),
 comparray2);
}

/* A necessary predefinition */
DB_CompArray NewDBScalarMult_(DB_Info info, DB_CompArray comparray,
      double factor);

/* A necessity for messy data. Sets any values in the comparray > 1 to 1 */
DB_CompArray NewDBShear_(DB_Info info, DB_CompArray comparray1)
{
  double *ca1_runner;
  unsigned long int i;

  ca1_runner = comparray1;

  for(i=0; i<info->numwords; i++) {
    if(*ca1_runner > 1) *ca1_runner = 1;
    ca1_runner++;
  }
}

DB_CompArray NewDBShear(DB_Info info, DB_CompArray comparray1)
{
  return NewDBShear_(info, NewDBCompArrayDup(info, comparray1));
}

/* Convolution functions */
DB_CompArray NewDBOneify_(DB_Info info, DB_CompArray comparray1)
{
  return NewDBScalarMult_(info,
  comparray1,
  1/NewDBIntegrateArray(info, comparray1));
}

DB_CompArray NewDBOneify(DB_Info info, DB_CompArray comparray1)
{
  return NewDBOneify_(info, NewDBCompArrayDup(info, comparray1));
}

/* I don't know FFTs, so this will be slow. */
DB_CompArray NewDBConvolve(DB_Info info, DB_CompArray comparray1,
 DB_CompArray comparray2)
{
  int j, k;
  DB_CompArray comparray3;

  comparray3 = NewDBAllocZeroArray(info);

  for(j=0; j<info->numwords; j++)
    for(k=0; k<info->numwords; k++)
      comparray3[j] += comparray1[  k   % info->numwords] *
       comparray2[(j-k) % info->numwords];

  return comparray3;
```

```
}

/* Likewise */
DB_CompArray NewDBCorrelate(DB_Info info, DB_CompArray comparray1,
  DB_CompArray comparray2)
{
  int i, j;
  DB_CompArray comparray3;

  comparray3 = NewDBAllocZeroArray(info);

  for(i=0; i<info->numwords; i++)
    for(j=0; j<info->numwords; j++)
      comparray3[i] += comparray1[  j   % info->numwords] *
       comparray2[(i+j) % info->numwords];

  return comparray3;
}


DB_CompArray NewDBScalarMult_(DB_Info info, DB_CompArray comparray,
      double factor)
{
  double *ca_runner;
  unsigned long int i;

  ca_runner = comparray;
  for(i=0; i<info->numwords; i++) *ca_runner++ *= factor;

  return(comparray);
}

DB_CompArray NewDBScalarMult(DB_Info info, DB_CompArray comparray,
     double factor)
{
  return NewDBScalarMult_(info,
     NewDBCompArrayDup(info, comparray),
  factor);
}

double findnormalizer(DB_Info info, DB_CompArray comparray)
{
  unsigned long int i;
  double hival=0, *ca_runner;

  ca_runner = comparray;
  for(i=0; i<info->numwords; i++) {
    if(*ca_runner > hival) hival = *ca_runner;
    ca_runner++;
  }

  return(hival == 0 ? hival : 1/hival);
}

DB_CompArray NewDBNormalizeArray_(DB_Info info, DB_CompArray comparray)
{
  return NewDBScalarMult_(info, comparray, findnormalizer(info, comparray));
}

DB_CompArray NewDBNormalizeArray(DB_Info info, DB_CompArray comparray)
{
  return NewDBNormalizeArray_(info, NewDBCompArrayDup(info, comparray));
}

/* Something USEFUL */

int NewDBWordExists(DB_Info info, char *wd)
{
  /*
  datum d;
  d.dptr = wd;
  d.dsize = strlen(wd)*sizeof(char) + 1;
```

```
  return gdbm_exists(info->index, d);
  */
  DB_RawArray kludge;

  kludge = NewDBLoadArray(info, wd);

  if(kludge) {
    NewDBZapRawArray(kludge);
    return(1);
  } else return(0);
}

/* Speedy routines */

double NewDBDifferenceFactor_s(DB_Info info, DB_CompArray comparray1,
     DB_CompArray comparray2)
{
  double result=0, *ca1_runner, *ca2_runner;
  unsigned long int i;

  ca1_runner = comparray1;
  ca2_runner = comparray2;
  for(i=0; i<info->numwords; i++)
    result += fabs(*ca1_runner++ - *ca2_runner++);
  return(result);
}

double NewDBDiffFactGiveUp_s(DB_Info info, DB_CompArray comparray1,
   DB_CompArray comparray2,
   double scoretobeat)
{
  double result=0, *ca1_runner, *ca2_runner;
  unsigned long int i;

  ca1_runner = comparray1;
  ca2_runner = comparray2;
  for(i=0; i<info->numwords; i++) {
    result += fabs(*ca1_runner++ - *ca2_runner++);
    if(result > scoretobeat) return(HUGE_VAL);
  }
  return(result);
}

/* File routines */

void NewDBSaveCompArray(DB_Info info, DB_CompArray comparray, char *path)
{
  FILE *savefile;
  unsigned long int size;

  UNLESS(savefile = fopen(path, "w")) file_die(path);
  size = fwrite(comparray, sizeof(double), info->numwords, savefile);
  UNLESS(size == info->numwords) {
    fprintf(stderr,
    "%s: NewDBlib error: The number of words saved to\n", progname);
    fprintf(stderr, "  %s\n", path);
    fprintf(stderr, "is %ld. It should have saved %ld words!\n",
    size, info->numwords);
  }
  fclose(savefile);
}

DB_CompArray NewDBLoadCompArray(DB_Info info, char *path)
{
  DB_CompArray newarray;
  unsigned long int size;
  FILE *loadfile;

  UNLESS(newarray = malloc(info->numwords*sizeof(double)))
    mem_die("a computed array loaded from a file");
  UNLESS(loadfile = fopen(path, "r")) file_die(path);
```

```
  size = fread(newarray, sizeof(double), info->numwords, loadfile);
  UNLESS(size == info->numwords) {
    fprintf(stderr,
    "%s: NewDBlib error: The number of words loaded from\n", progname);
    fprintf(stderr, "  %s\n", path);
    fprintf(stderr, "is %ld. It should have saved %ld words!\n",
    size, info->numwords);
  }
  fclose(loadfile);

  return(newarray);
}

/* Interoperability functions */
double *NewDBCompArrayToDouble_(DB_Info info, DB_CompArray comparray)
{
  return comparray;
}

double *NewDBCompArrayToDouble(DB_Info info, DB_CompArray comparray)
{
  return NewDBCompArrayToDouble_(info, NewDBCompArrayDup(info, comparray));
}

float *NewDBCompArrayToFloat(DB_Info info, DB_CompArray comparray)
{
  double *d;
  float *FloatArray, *f;
  int i;

  UNLESS(FloatArray = malloc(info->numwords * sizeof(float)))
    mem_die("an array of floats");

  d = comparray;
  f = FloatArray;
  for(i=0; i<info->numwords; i++) *f++ = *d++;

  return FloatArray;
}

DB_CompArray NewDBFloatToCompArray(DB_Info info, float *array)
{
  DB_CompArray comparray;
  int i;

  comparray = NewDBAllocZeroArray(info);
  for(i=0; i<info->numwords; i++) comparray[i] = array[i];

  return comparray;
}


/* Neural network helper */
#define DEVIANCE 0.000001
DB_CompArray NewDBNeuralizeCompArray_(DB_Info info, DB_CompArray comparray)
{
  double *d, tmp = 1 - (2*DEVIANCE);
  int i;

  d = comparray;
  for(i=0; i<info->numwords; i++) {
    if(*d > tmp) *d = tmp;
    *d += DEVIANCE;
    d++;
  }
  return comparray;
}

DB_CompArray NewDBNeuralizeCompArray(DB_Info info, DB_CompArray comparray)
{
  return NewDBNeuralizeCompArray_(info, NewDBCompArrayDup(info, comparray));
}
```

Acknowledgement of SWIG

SWIG, the Simplified Wrapper and Interface Generator, was used extensively throughout Summer 2000. SWIG takes the interfaces to C/C++ libraries defined in ordinary header files and creates wrappers that permit you to call C/C++ routines in many popular scripting languages. This leads naturally to a design approach in which established, fast components are written in a lower-level language (viz. NewDB.c) while less speed-critical, more experimental code is rapidly prototyped and tested in a high-level language. Without SWIG, the broad range of investigated topics in the Summer 2000 research might never have been achieved.

See also `http://www.swig.org/` and:

David M. Beazley, "SWIG: An Easy To Use Tool For Integrating Scripting Languages with C and C++". *Proceedings of the USENIX Fourth Annual Tcl/Tk Workshop*, (1996)

synonyms.pl

This simple Perl script uses the SWIG-generated interface to NewDB to find the twenty nearest matches to a word specified on the command line. The brevity of this program is a testiment to the efficiency of using higher-level languages in conjunction with lower-level libraries.

The "train of thought" program from section 2.4 is a modified version of this program: it contains a loop around the main loop below in which it picks at random a match from the twenty nearest matches to a word, searches for its twenty nearest matches, picks a match from the resulting choices, and so on.

```
#!/usr/local/bin/perl -w

use strict;
use lib '/local/tom/00/lib/perl5/site_perl/';
use NewDB;

unless(@ARGV == 1) { die "Usage: $0 word"; }

my $datapath = "/local/tom/00/db-mini";
my $numsynms = 20;
my $target = $ARGV[0];

opendir(DATADIR, $datapath) || die "$0: Can't stat dir $datapath";
my @words = grep { !/^\./ && !/^[A-Z]+/ && !/^$target$/ } readdir(DATADIR);
closedir DATADIR;

# init bestwords list

my @bestwords = (['Error!', 99999]) x $numsynms;

# Preliminary database stuff

my $info = NewDB::NewDBinit($datapath);

my $raw = NewDB::NewDBLoadArray($info, $target)
         || die "$0: Word $target does not exist in the database.";
my $wdcmp = NewDB::NewDBComputeArray($info, $raw);
NewDB::NewDBZapRawArray($raw);

# The main loop

foreach my $cmpwd (@words) {
  my $raw = NewDB::NewDBLoadArray($info, $cmpwd) || die "$0: OOPS! $cmpwd";
  my $cmpcmp = NewDB::NewDBComputeArray($info, $raw);
  NewDB::NewDBZapRawArray($raw);

  my $score = NewDB::NewDBDifferenceFactor_s($info, $wdcmp, $cmpcmp);
  NewDB::NewDBZapCompArray($cmpcmp);

  if($score < $bestwords[0][1]) {
    $bestwords[0] = [$cmpwd, $score];
    print "Insertion: $score ($cmpwd)\n";
    @bestwords = sort { $b->[1] <=> $a->[1] } @bestwords;
  }
}

# Cleanup

NewDB::NewDBZapCompArray($wdcmp);
NewDB::NewDBclose($info);

print "The $numsynms words most closely matching $target: \n";
foreach my $entry (@bestwords) { print "\t$entry->[0]:\t($entry->[1])\n"; }
```

clustmeister.c

This code performed the cluster analysis on the words database, printing out the cluster tree in a format employing nested parentheses. Later scripts (not included in this appendix) converted the tree format into raw Postscript code for Figure 8.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>
#include <math.h>

#include "NewDB.h"

#define UNLESS(item)   if(!( (item) ))
#define DBPATH "/local/tom/db-provisional/"
#define MAX_WD_LENGTH 40

typedef struct _bintree_rec {
  struct _bintree_rec *left, *right;
  char name[MAX_WD_LENGTH];
} bintree_rec, *bintree;

char *progname;
DB_Info info;
float **HugeAdjacencyMatrix;
bintree *Current=NULL;

int numkeys=0, num_non_null;
int ClusterFileSerial=0;
FILE *outfile;

#ifdef _NEWDB_H_
extern void die(char *why);
#else
void die(char *why)
{
  fprintf(stderr, "%s: %s\n", progname, why);
  exit(-1);
}
#endif

void InitCurrent(void)
{
  DIR *myDIR;
  struct dirent *myEnt;
  bintree leaf;
  int i = 0, j;

  UNLESS(myDIR = opendir(DBPATH)) die("Please correct DBPATH in source code");
  while((myEnt = readdir(myDIR))) {
    /* Skip directories, index files kludge */
    if((*myEnt->d_name == '.') || (*myEnt->d_name == toupper(*myEnt->d_name)))
      continue;

    UNLESS(leaf = malloc(sizeof(bintree_rec))) die("Out of LEAF memory");
    leaf->left = leaf->right = NULL;
    strncpy(leaf->name, myEnt->d_name, MAX_WD_LENGTH);
    leaf->name[MAX_WD_LENGTH-1] = '\0';

    UNLESS(Current = realloc(Current, sizeof(bintree)*++numkeys))
      die("Couldn't reallocate memory for Current array");
    Current[numkeys-1] = leaf;
  }
  closedir(myDIR);
  num_non_null = numkeys;
  printf("'Current' array initialized: %d entries.\n", numkeys);

  UNLESS(HugeAdjacencyMatrix = malloc(sizeof(float*)*numkeys))
```

```
    die("HugeAdjacencyMatrix malloc failed.");

  for(i=0; i<numkeys-1; i++) {
    UNLESS(HugeAdjacencyMatrix[i] =
   malloc(sizeof(float)*(numkeys - i+1)))
      die("HugeAdjacencyMatrix submalloc failed.");

    /* Realign the pointers so we can address the HAM like a normal 2D array */
    /* Heh heh. */
    HugeAdjacencyMatrix[i] -= i+1;
  }
  puts("HugeAdjacencyMatrix malloc'd and pointers aligned.");
}

DB_CompArray LoadCompArray(char *word)
{
  DB_RawArray raw;
  DB_CompArray target;

  if(isdigit((int) *word)) {
    UNLESS(target = NewDBLoadCompArray(info, word)) {
      fprintf(stderr, "Can't find computed array %s\n", word);
      die("I've lost track of my head.");
    }

    return(target);
  }
  else {
    UNLESS(raw = NewDBLoadArray(info, word)) {
      fprintf(stderr, "Can't find wordfile for %s\n", word);
      die("Something went awry in InitCurrent");
    }

    target = NewDBComputeArray(info, raw);
    NewDBZapRawArray(raw);

    return(target);
  }
}

double Compare(DB_CompArray ca1, int d2)
{
  double result;
  DB_CompArray ca2;

  ca2 = LoadCompArray(Current[d2]->name);

  result = NewDBDifferenceFactor_s(info, ca1, ca2);
  NewDBZapCompArray(ca2);

  return(result);
}

void LoadUpAdjacencyMatrix(void)
{
  int i, j;
  float score;
  DB_CompArray base;
  char roller[] = {'/', '-', '\\', '|'};

  for(i=0; i<numkeys-1; i++) {
    base = LoadCompArray(Current[i]->name);
    for(j=i+1; j<numkeys; j++) {
      score = Compare(base, j);
      /* space-saving trick */
      HugeAdjacencyMatrix[i][j] = score;
      if(j % 25 == 0) {
printf("\b%c", roller[j%4]);
fflush(stdout);
      }
    }
    NewDBZapCompArray(base);
```

```
    printf("\b.\\");
    fflush(stdout);
  }

  puts("\b!");
  puts("Adjacency matrix loaded.");
}

bintree Consolidate(int d1, int d2)
{
  int i;
  float score;
  DB_CompArray ca1, ca2, base;
  char filename[MAX_WD_LENGTH];
  bintree node;

  printf("Consolidating node %s and %s into node %d...",
 Current[d1]->name, Current[d2]->name, ClusterFileSerial);
  fflush(stdout);

  ca1 = LoadCompArray(Current[d1]->name);
  ca2 = LoadCompArray(Current[d2]->name);

  /* compute the average */
  NewDBSumArray_(info, ca1, ca2);
  NewDBScalarMult_(info, ca1, 0.5);

  sprintf(filename, "%d", ClusterFileSerial++);
  NewDBSaveCompArray(info, ca1, filename);
  NewDBZapCompArray(ca2);

  /* And now to make the link. */
  UNLESS(node = malloc(sizeof(bintree_rec))) die("bintree node alloc failure");

  strcpy(node->name, filename); /* this is safe here. */
  node->left = Current[d1];
  node->right = Current[d2];
  Current[d1] = node;
  Current[d2] = NULL;

  /* Clean up the HugeAdjacencyMatrix */
  /* The "+ d2+1" is to "unalign" the pointer for freeing. */
  if(HugeAdjacencyMatrix[d2]==NULL) die("FAULT!");
  //free(HugeAdjacencyMatrix[d2] + d2+1);
  HugeAdjacencyMatrix[d2] = NULL;

  /* Compute new entries for new array */
  /* ca1 is generated above */
  for(i=0; i<d1; i++) {
    UNLESS(HugeAdjacencyMatrix[i]) continue;
    score = Compare(ca1, i);
    HugeAdjacencyMatrix[i][d1] = score;
  }
  for(i=d1+1; i<numkeys; i++) {
    UNLESS(HugeAdjacencyMatrix[i]) continue;
    score = Compare(ca1, i);
    HugeAdjacencyMatrix[d1][i] = score;
  }

  /* Finally. */
  NewDBZapCompArray(ca1);
  num_non_null--;
  puts("Done");

  return(node);
}

bintree DoCluster(void)
{
  int i, j, best_i, best_j;
  float score, bestscore;
  bintree top=NULL;
```

```
  while(num_non_null > 1) {
    bestscore = HUGE_VAL;
    best_i = best_j = 0;

    for(i=0; i<numkeys-1; i++) {
      UNLESS(HugeAdjacencyMatrix[i]) continue;

      for(j=i+1; j<numkeys; j++) {
        UNLESS(HugeAdjacencyMatrix[j]) continue;
score = HugeAdjacencyMatrix[i][j];
if(score < bestscore) {
  bestscore = score;
  best_i = i;
  best_j = j;
}
      }
    }

    top = Consolidate(best_i, best_j);
  }

  return(top);
}

void Traverse(bintree node)
{
  fprintf(outfile, "(%s ", node->name);

  if(node->left) Traverse(node->left);
    else fprintf(outfile, "NULL ");

  if(node->right) Traverse(node->right);
    else fprintf(outfile, "NULL");

  fprintf(outfile, ")");
  free(node);
}

int main(int argc, char **argv)
{
  bintree root;

  progname = *argv;

  info = NewDBinit(DBPATH);
  InitCurrent();
  LoadUpAdjacencyMatrix();
  root = DoCluster();
  NewDBclose(info);

  UNLESS(outfile = fopen("clustmeister.out", "w")) outfile = stdout;
  printf("Dumping cluster tree to disk..."); fflush(stdout);
  Traverse(root);
  fclose(outfile);
  puts(" done. The end.");

  return(0);
}
```

McGwire.cc and the Slugger suite of neural network programs

The programs used for the neural network portion of the Summer 2000 research (Section 2.6) are collectively referred to as the Slugger suite by dint of their names. The source code for the entire suite is too voluminous to include here, so only McGwire, the simple neural network training program, is provided. McGwire has little to set it apart from other programs that train and save simple feed-forward, backpropagation-trained neural networks except that it will automatically load word representations on its own (as opposed to requiring that the data be dumped in a training file beforehand).

Other Slugger suite programs include Sosa, which evaluated networks made by McGwire, and Ruth and Aaron, which concerned a small side project involving creating and taking aparm RAAM representations of sequences of word representations. Little formal work was done with the latter two programs.

All Slugger suite programs made extensive use of Daniel Franklin's free libneural neural network library.[8]

```
// McGwire.cc - the neural network training program - in -*- C++ -*-
// Necessary C++ stuff
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <strstream>
#include <algorithm>

// A useful C thing
#include <string.h>

// The DYDEX library
extern "C" {
#include <NewDB.h>
}

// The neural network library
#include <nnwork.h>

// McConfig configuration class header
#include "McConfig.h"

// Here's our favorite #define
#define UNLESS(item) if(!( (item) ))

// Functions
void LoadWord(DB_Info info, const string& word, float **ArrayPtr);
void TrainNet(McConfig& myConf, nnwork& TheNet);

// Globals
char *progname;

// Now we should be ready to go.
int main(int argc, char **argv)
{
  // set progname, for NewDB's benefit
  progname = *argv;

  // check argument sanity
  UNLESS(argc == 2) die("usage: McGwire project-file");

  // initialize config object and print discovered information
  McConfig myConf(argv[1]);
  myConf.PrintCfgInfo();

  // initialize the neural net
  nnwork TheNet(myConf.getNumInput(),
myConf.getNumHidden(),
myConf.getNumOutput());

  // is the net pretrained?
```

[8] http://ieee.uow.edu.au/~daniel/software/libneural/

```
  if(myConf.getFlags() & F_PRETRAINED) {
    cout << "Loading pretrained network..." << endl;
    int l = myConf.getName().length();
    char *filename = new char[l+5];

    myConf.getName().copy(filename, l, 0);
    strcpy(&(filename[l]), ".nnw");

    UNLESS(TheNet.load(filename)) die("Couldn't load pretrained network");

    // check for the same network characteristics
    UNLESS((TheNet.get_layersize(INPUT)  == myConf.getNumInput()) &&
   (TheNet.get_layersize(HIDDEN) == myConf.getNumHidden()) &&
   (TheNet.get_layersize(OUTPUT) == myConf.getNumOutput()))
      die("The pretrained net doesn't match the specified parameters");
  }

  cout << "=> Neural network initialized." << endl;

  // train the neural net
  TrainNet(myConf, TheNet);

  // save the network
  string tmp = myConf.getName() + ".nnw";
  int l = tmp.length();
  char *savename = new char[l + 1];
  tmp.copy(savename, l, 0);
  savename[l] = '\0';
  TheNet.save(savename);
  cout << "=> Neural network saved as " << tmp << endl;

  // that's it! Good night!
  return(0);
}

// Master of the TRAIN
void TrainNet(McConfig& myConf, nnwork& TheNet)
{
  DB_Info info = myConf.getDBinfo();

  // Memory acquisition
  float *Inputs   = new float[myConf.getNumInput()];
  float *Outputs  = new float[myConf.getNumOutput()];
  //float *Hiddens  = new float[myConf.getNumHidden()];
  float *LastOuts = new float[myConf.getNumOutput()];

  UNLESS(Inputs && Outputs && /* Hiddens && */ LastOuts)
    die("Couldn't allocate memory for network data storage");

  cout << "=> Training network..." << endl;

  vector< int > InFmt = myConf.getInFmt();
  vector< int > OutFmt = myConf.getOutFmt();
  vector< string > Trials = myConf.getTrials();

  // Begin training
  for(int iters=0; iters<myConf.getIterations(); iters++) {
    // randomize if specified
    if(myConf.getFlags() & F_RANDOM)
      random_shuffle(Trials.begin(), Trials.end());

    // Zero the data in LastOuts and LastHiddens
    for(int i=0; i<myConf.getNumInput();  i++) Inputs[i]  = 0.0;
    //for(int i=0; i<myConf.getNumHidden(); i++) Hiddens[i] = 0.0;

    // now traverse the trials
    vector< string >::const_iterator trial;
    vector< int >::const_iterator i;
    for(trial=Trials.begin(); trial != Trials.end(); trial++) {
      float *InRunner = Inputs;
      float *OutRunner = Outputs;
```

```
      // Check for reset first, and zap the hidden layer if it shows up.
      UNLESS(trial->compare("!reset")) { TheNet.zap_hidden(); continue; }
      // Any others mean nothing to the training program
      if((*trial)[0] == '!') continue;

      // If the Flag: Flaunt is on, say what we're doing.
      if(myConf.getFlags() & F_FLAUNT) cout << '\t' << *trial << endl;

      // All clear. Continue with the training.
      istrstream trS(trial->data(), trial->length());

      // Set up the input vector
      for(i=InFmt.begin(); i != InFmt.end(); i++) {
switch(*i) {
  case c_word: {
    string wd;
    UNLESS(trS >> wd) {
      cerr << '\'' << trial <<  "' is missing a needed word." << endl;
      die("Invalid network input specification");
    }
    LoadWord(info, wd, &InRunner);
  } break;
  case c_scalar: {
    UNLESS(trS >> *InRunner++) {
      cerr << '\'' << trial <<  "' is missing a needed scalar." << endl;
      die("Invalid network input specification");
    }
  } break;
  case c_lasthidden: {
    TheNet.get_hidden(InRunner);
    InRunner += myConf.getNumHidden();
  } break;
  case c_lastout: {
    float *LOutRunner = LastOuts;
    for(int j=0; j<myConf.getNumOutput(); j++)
      *InRunner++ = *LOutRunner++;
  } break;
  // There should be no other cases!
}
      }

      // Set up the output vector
      for(i=OutFmt.begin(); i != OutFmt.end(); i++) {
switch(*i) {
  case c_word: {
    string wd;
    UNLESS(trS >> wd) {
      cerr << '\'' << trial <<  "' is missing a needed word." << endl;
      die("Invalid network output specification");
    }
    LoadWord(info, wd, &OutRunner);
  } break;
  case c_scalar: {
    UNLESS(trS >> *OutRunner++) {
      cerr << '\'' << trial <<  "' is missing a needed scalar." << endl;
      die("Invalid network output specification");
    }
  } break;
  case c_lasthidden: {
    TheNet.get_hidden(OutRunner);
    OutRunner += myConf.getNumHidden();
  } break;
  case c_input: {
    float *InptRunner = Inputs;
    for(int j=0; j<myConf.getNumOutput(); j++)
      *OutRunner++ = *InptRunner++;
  } break;
  // There should be no other cases!
}
      }

      // Train the network
```

```
      TheNet.train(Inputs, Outputs, myConf.getMaxErr(), myConf.getLearnRate());

      // Copy the network hidden layer activation and output activation
      // to Hiddens and LastOuts.
      // This cannot be done without a modification to libneural.
      // We won't do it yet.
    }

    cout << "Training round " << iters+1 << " complete." << endl;
  }

  cout << "=> Training complete." << endl;

  delete Inputs;
  delete Outputs;
  //delete Hiddens;
  delete LastOuts;
}

void LoadWord(DB_Info info, const string& word, float **ArrayPtr)
{
  int l = word.length();
  char *cword = new char[l+1];
  word.copy(cword, l, 0);
  cword[l] = '\0';

  DB_CompArray ca;

  if(DB_RawArray raw = NewDBLoadArray(info, cword)) {
    ca = NewDBComputeArray(info, raw);
    NewDBZapRawArray(raw);
  }
  else ca = NewDBLoadCompArray(info, cword);

  NewDBNeuralizeCompArray_(info, ca);
  float *f, *fptr;
  f = fptr = NewDBCompArrayToFloat(info, ca);

  // Copy the floatized word array to the inputs
  // DANGER - direct access to info struct!
  for(unsigned int i=0; i<info->numwords; i++) *(*ArrayPtr)++ = *fptr++;

  free(f);
  NewDBZapCompArray(ca);
  delete cword;
}
```

A.2 Summer 2001 research code

This section of Appendix A will present all of the Summer 2001 code library, known as libdydex. It will then present Robot.cc, the program which uses it to control the AmigoBOT robot.

Relation.h

This header file describes the data structure for relations. The id member of the structure is not currently used.

```
/*
 * Relations are measures of relatedness between LinkPaks. Accordingly,
 * they are represented by simple structures.
 */
#ifndef _RELATION_H_
#define _RELATION_H_

class LinkPak; // predefinition - see LinkPak.h

typedef struct _Relation {
    LinkPak* TargetLinkPak; // the target of this relation
    double Strength; // the strength of this relation
    unsigned long id; // an ID number for this relation
} Relation;

#endif
```

LinkPak.h

This header file describes the LinkPak object, which implements graph nodes.

```
/*
 * LinkPaks are the fundamental units of the DYDEX system: collections of
 * Relations approximating the relatedness of this LinkPak with others.
 * A hashtable using LinkPak references as keys keeps them all together.
 */
#ifndef _LINK_PAK_H_
#define _LINK_PAK_H_

#include <iostream>
#include <string>
#include <hash_map.h>
//#include <map>
#include <list>

#include "Relation.h"

// The default name for LinkPaks
#define DEF_LP_NAME "Untitled concept"

// A note about hash_maps
// The current state of hash maps allows you to use only certain sanctioned
// types as hash keys. Pointers are not among them. I'll try casting pointers
// to longs for the sake of hash_map and then see if it performs well. This
// conversion will naturally be architecture dependent, so porters may have
// to change it. Porters without hash_map on their systems may substitute
// the appropriate commented lines above and below this note.
#define PAK_KEY_TYPE long
#define PAK_KEY_CAST (PAK_KEY_TYPE)
#define PAK_KEY_UNCAST (LinkPak*)
//#define PAK_KEY_TYPE LinkPak*
//#define PAK_KEY_CAST
//#define PAK_KEY_UNCAST

typedef hash_map<PAK_KEY_TYPE, Relation> PakMap;
//typedef map<PAK_KEY_TYPE, Relation> PakMap;

using namespace std;

class LinkPak {
  friend class LPCBase1; // declare all the compare classes
  friend class LPCBase2; // as Friends, so that they have
  friend class LPCBase3; // access to the Pak hashtable
  friend class LPCBase4;
  friend class LPCRecursive;
  friend class LPCRecursiveAdd;
  private:
    static unsigned long serial; // used to create unique IDs
    string Name; // the name of this LinkPak
    unsigned long ID; // and a unique numerical ID

    class LPFindID; // these classes are helper functors
    class LPFindStr; // for the find functions.
  protected:
    static unsigned long LRUctr; // used to set LRUs
    unsigned long LRU; // data for the LRU purging algorithm
    PakMap Pak; // the collection of Relations
  public:
    LinkPak(string); // The constructor
    unsigned long id() const; // the ID of this LinkPak
    unsigned long lru() const; // the LRU value of this LinkPak
    string name() const; // The name of this LinkPak
    int size() const; // # of Relations in this LinkPak
    bool operator==(LinkPak&) const; // equality operator for LinkPaks
    virtual bool insert(Relation&); // Insert Relation into LinkPak
    virtual bool insert(LinkPak*, double, unsigned long); // similar

    // the following functions are kludges so that the Interface class can
    // use insertForever and still be general to all LinkPaks. Oh well.
    // See LinkPak.cc for more information.
```

```
    bool insertForever(Relation&);
    bool insertForever(LinkPak*, double, unsigned long);

    Relation* find(LinkPak*); // Return Relation pointing to T
    Relation* find(unsigned long); // Return Relation to LinkPak id ID
    Relation* find(string&); // Return Relation to LinkPak "name"

    void print(ostream&); // prints out this LinkPak
    list<Relation> Contents(); // Returns all Relations in this LinkPak

    virtual ~LinkPak() { } // virtual destructor
};
#endif
```

LinkPak.cc

This file contains the code for the LinkPak class, which implements graph nodes.

```
/*
 * LinkPaks are the fundamental units of the DYDEX system: collections of
 * Relations approximating the relatedness of this LinkPak with others.
 * A hashtable using LinkPak references as keys keeps them all together.
 */

#include <string>
#include <iostream>
#include <hash_map.h>
//#include <map>
#include <list>

#include "LinkPak.h"
#include "Relation.h"
#include "LinkPakCompare.h"

#define UNLESS(item) if(!( (item) ))

using namespace std;

// First, initialize the serial number for the IDs of LinkPaks:
unsigned long LinkPak::serial = 0;
// Also initialize the LRU counter:
unsigned long LinkPak::LRUctr = 0;

// The constructor
LinkPak::LinkPak(string name=DEF_LP_NAME)
{
  Name = name;
  ID = serial++;
  LRU = LRUctr++;
}

// the ID of this LinkPak
inline unsigned long LinkPak::id(void) const
{
  return ID;
}

// the LRU value of this LinkPak
// gcc 3 didn't like this to be inline.
unsigned long LinkPak::lru(void) const
{
  return LRU;
}

// the name of this LinkPak
inline string LinkPak::name(void) const
{
  return Name;
}

// # of Relations in this LinkPak
int LinkPak::size(void) const
{
  return Pak.size();
}

// Insert Relation into LinkPak
bool LinkPak::insert(Relation& r)
{
  LRU = LRUctr++;
  return Pak.insert(make_pair(PAK_KEY_CAST r.TargetLinkPak, r)).second;
}

// Create a new Relation with this data and insert into LinkPak
bool LinkPak::insert(LinkPak* T, double S, unsigned long ID)
{
  Relation r;
```

```
  LRU = LRUctr++;

  r.TargetLinkPak = T;
  r.Strength = S;
  r.id = ID;

  return Pak.insert(make_pair(PAK_KEY_CAST T, r)).second;
};

// Note - These have been transplanted from LimitedLinkPak.cc for
// interoperability reasons.
// The insertForever function behaves more like the traditional insert function
// in LinkPaks - it adds a value never to be pushed out from the system if
// a stronger Relation is added later.
bool LinkPak::insertForever(Relation& r)
{
  return LinkPak::insert(r);
}

// Ditto for this one
bool LinkPak::insertForever(LinkPak* l, double s, unsigned long id)
{
  return LinkPak::insert(l, s, id);
}

// The equality operator for LinkPaks. Compares serial numbers.
bool LinkPak::operator==(LinkPak& l) const
{
  return ID == l.id();
}

// Return Relation pointing to T, or null if none exists.
Relation* LinkPak::find(LinkPak* T)
{
  PakMap::iterator i = Pak.find(PAK_KEY_CAST T);

  return i == Pak.end() ? NULL : &i->second;
}

// Return Relation with id ID or NULL if none exists. Accomplishing this
// requires the following functor first:
class LinkPak::LPFindID {
  private:
    unsigned long Target;
  public:
    LPFindID(unsigned long T) { Target = T; }
    bool operator() (pair<const PAK_KEY_TYPE, const Relation> p) const {
      return (PAK_KEY_UNCAST (p.first))->id() == Target;
    }
};

// And now the function itself
Relation* LinkPak::find(unsigned long ID)
{
  PakMap::iterator i = find_if(Pak.begin(), Pak.end(), LinkPak::LPFindID(ID));

  return i == Pak.end() ? NULL : &i->second;
}

// Return Relation with name "name" or NULL if none exists. Accomplishing
// this requires the following functor first:
class LinkPak::LPFindStr {
  private:
    string Target;
  public:
    LPFindStr(string T) { Target = T; }
    bool operator() (pair<const PAK_KEY_TYPE, const Relation> p) const {
      return (PAK_KEY_UNCAST (p.first))->name() == Target;
    }
};
```

```
// And now the function itself
Relation* LinkPak::find(string& name)
{
  PakMap::iterator i= find_if(Pak.begin(), Pak.end(), LinkPak::LPFindStr(name));

  return i == Pak.end() ? NULL : &i->second;
}

// Finally, a function that prints a string representation of this LinkPak
void LinkPak::print(ostream& out)
{
  out << "# LinkPak " << ID << " (" << Name << ") <LRU: " << LRU << ">\n";

  for(PakMap::iterator i = Pak.begin(); i != Pak.end(); i++)
out << "# " << (PAK_KEY_UNCAST i->first)->id() << " ("
    << (PAK_KEY_UNCAST i->first)->name() << ")\t- "
    << i->second.Strength << '\n';
}

// This function returns all the Relations in this LinkPak
list<Relation> LinkPak::Contents(void)
{
  list<Relation> l;

  for(PakMap::iterator i = Pak.begin(); i != Pak.end(); i++)
    l.push_back(i->second);

  return l;
}
```

LimitedLinkPak.h

A LimitedLinkPak is a graph node like the LinkPak. However, it limits the number of relations it contains to a prescribed value. When a new relation is inserted into a LimitedLinkPak, the insertion is unsuccessful unless the relation is stronger than any of the relations the LimitedLinkPak already contains.

```
/*
 * A LimitedLinkPak is a LinkPak that limits the Relations it contains to
 * a prescribed value n. When a new Relation is inserted into a LimitedLinkPak,
 * it isn't actually inserted unless it's one of the n highest Relations
 * ever submitted for insertion.
 */
#ifndef _LIMITED_LINK_PAK_H_
#define _LIMITED_LINK_PAK_H_

#include <string>
#include <queue>
#include <vector>
#include "Relation.h"
#include "LinkPak.h"

// The LLPRelGT class sorts Relations for the priority queue in all
// LimitedLinkPaks
class LLPRelGT {
  public:
    bool operator()(Relation, Relation);
};

typedef priority_queue<Relation, vector<Relation>, LLPRelGT> LLPqueue;

class LimitedLinkPak : public LinkPak {
  private:
    unsigned int Limit;
    double worstbest;

    LLPqueue Best;
  public:
    LimitedLinkPak(string, unsigned int); // The constructor
    bool insert(Relation&); // Insert Relation into LinkPak
    bool insert(LinkPak*, double, unsigned long); // similar
    // insertForever functions have been moved to LinkPak.*
};
#endif
```

LimitedLinkPak.cc

This code implements the LimitedLinkPak class, which is described on the previous page.

```
/*
 * A LimitedLinkPak is a LinkPak that limits the Relations it contains to
 * a prescribed value n. When a new Relation is inserted into a LimitedLinkPak,
 * it isn't actually inserted unless it's one of the n highest Relations
 * ever submitted for insertion.
 */

#include <iostream>

#include <string>
#include <queue>

#include "Relation.h"
#include "LinkPak.h"
#include "LimitedLinkPak.h"

// The LLPRelGT class sorts Relations for the priority queue in all
// LimitedLinkPaks. In particular, it does so with the () operator.
bool LLPRelGT::operator()(Relation r1, Relation r2) {
  return r1.Strength > r2.Strength;
}

LimitedLinkPak::LimitedLinkPak(string n, unsigned int l) : LinkPak(n), Limit(l)
{
  worstbest = 0;
}

// The insert function for LimitedLinkPaks differs from that of LinkPaks in
// that if a Relation with a lower Strength value than any already stored
// in a LimitedLinkPak is put in for insertion, the operation fails. This
// ensures that LimitedLinkPaks contain only the strongest Relations.
bool LimitedLinkPak::insert(Relation& r)
{
  if(Limit == 0) return false; // believe it or not, this makes sense somewhere

  if(r.Strength < worstbest) return false;
  // SLOWDOWN - FIX OR REMOVE?
  //if(Pak.find(PAK_KEY_CAST (r.TargetLinkPak)) != Pak.end()) return false;
  PakMap::iterator i = Pak.find(PAK_KEY_CAST (r.TargetLinkPak));
  if(i != Pak.end())
    if(r.Strength > i->second.Strength) {
      // update LRU
      LRU = LRUctr++;

      i->second.Strength = r.Strength;
      // Now we have to reshuffle the deck...
      queue<Relation> q;
      while(Best.top().TargetLinkPak != r.TargetLinkPak) {
q.push(Best.top());
Best.pop();
      }
      Best.pop();
      Best.push(i->second);
      while(q.size() > 0) {
        Best.push(q.front());
q.pop();
      }
      return true;
    }
    else return false;

  // this chunk is a bit elliptical for the sake of speed...
  if(Best.size() == Limit) {
    Pak.erase(Pak.find(PAK_KEY_CAST (Best.top().TargetLinkPak)));
    Best.pop();
    Pak.insert(make_pair(PAK_KEY_CAST r.TargetLinkPak, r));
    Best.push(r);
    worstbest = Best.top().Strength;
  }
```

```
  else {
    Pak.insert(make_pair(PAK_KEY_CAST r.TargetLinkPak, r));
    Best.push(r);
    if(Best.size() == Limit) worstbest = Best.top().Strength;
  }

  LRU = LRUctr++;
  return true;
}

// This insert function does the same thing as the last one - it just allows
// the user to enter the contents of a relation rather than make on on their
// own.
bool LimitedLinkPak::insert(LinkPak* T, double S, unsigned long ID)
{
  Relation r;

  r.TargetLinkPak = T;
  r.Strength = S;
  r.id = ID;

  return insert(r);
}

// The insertForever functions here are now in LinkPak.cc
// All this to satisfy Interface
```

LinkPakArchive.h

A LinkPakArchive is a container for LinkPaks. Used in the idiomatic way (i.e. one per program, with all LinkPaks stored within), a LinkPakArchive object can be thought of as the graph itself. The LinkPakArchive contains various functions for retrieving LinkPaks, including FindNearestNTo, which, supplied with an object that compares LinkPaks and a target LinkPak, will find a specified number of near matches to that LinkPak in the archive.

```
/*
 * A LinkPakArchive is an object that manages a large number of LinkPaks.
 * Besides just a listing, it contains various member functions that search
 * through the LinkPak and so forth.
 */
#ifndef _LINK_PAK_ARCHIVE_H_
#define _LINK_PAK_ARCHIVE_H_

#include <string>
#include <list>
#include <hash_map.h>// Please see "A note about hash_maps" in LinkPak.h!
//#include <map>

#include "LinkPak.h"
#include "LinkPakCompare.h"

#define DEF_LA_NAME "Untitled archive"

typedef hash_map<unsigned long, LinkPak*> ArcMap;
//typedef map<unsigned long, LinkPak*> ArcMap;

class LinkPakArchive {
  private:
    string Name; // the name of this LinkPakArchive
    ArcMap Archive; // the card catalog, so to speak

  public:
    LinkPakArchive(string);
    int size(); // size of the archive
    bool insert(LinkPak*); // insertion, naturally
    LinkPak* remove(unsigned long); // removal, naturally
    LinkPak* find(unsigned long); // finds a particular LinkPak

    // These search functions find a prescribed number of LinkPaks most
    // similar to the LinkPak specified. The user must also supply a
    // premade LinkPakCompare object for the routines to perform the search.
    list<Relation> FindNearestNTo(unsigned int, unsigned long, LinkPakCompare&);
    list<Relation> FindNearestNTo(unsigned int, LinkPak*, LinkPakCompare&);

    void print(ostream&); // prints out this LinkPak
    list<LinkPak*> Contents(void); // a list of this archive's contents
};
#endif
```

LinkPakArchive.cc

This code implements the LinkPakArchive class, which is described on the previous page.

```
/*
 * A LinkPakArchive is an object that manages a large number of LinkPaks.
 * Besides just a listing, it contains various member functions that search
 * through the LinkPak and so forth.
 */

#include <string>
#include <cstring>
#include <list>
#include <queue>
#include <hash_map.h>     // Please see "A note about hash_maps" in LinkPak.h!
//#include <map>
#include <math.h>

#ifdef DEBUG
#include <iostream>
#endif

#include "LinkPak.h"
#include "LimitedLinkPak.h" // Used for LLPRelGT class
#include "LinkPakArchive.h"

#define UNLESS(item) if(!( (item) ))

using namespace std;

// The constructor
LinkPakArchive::LinkPakArchive(string name=DEF_LA_NAME)
{
  Name = name;
}

int LinkPakArchive::size(void)
{
  return Archive.size();
}

// The insertion function
bool LinkPakArchive::insert(LinkPak* l)
{
  return Archive.insert(make_pair(l->id(), l)).second;
}

// The removal function
LinkPak* LinkPakArchive::remove(unsigned long pakid)
{
  ArcMap::iterator p = Archive.find(pakid);
  if(p == Archive.end()) return NULL;

  LinkPak* deadpak = p->second;
  Archive.erase(p);
  return deadpak;
}

// This function simply finds a LinkPak with the specified ID
LinkPak* LinkPakArchive::find(unsigned long pakid)
{
  ArcMap::iterator p = Archive.find(pakid);
  if(p == Archive.end()) return NULL;

  return p->second;
}

// These search functions find a prescribed number of LinkPaks most
// similar to the LinkPak specified. The user must also supply a
// premade LinkPakCompare object for the routines to perform the search.
inline list<Relation> LinkPakArchive::FindNearestNTo(unsigned int count,
unsigned long pakid,
LinkPakCompare& C)
```

```
{
  return FindNearestNTo(count, find(pakid), C);
}

list<Relation> LinkPakArchive::FindNearestNTo(unsigned int count, LinkPak* T,
LinkPakCompare& C)
{
  LLPqueue Best;
  double worstbest = 0;
  Relation r;

  r.id = 0;

  for(ArcMap::iterator i = Archive.begin(); i != Archive.end(); i++) {
    if(*i->second == *T) continue; // let's not be narcissistic...

    r.Strength = C.Compare(*T, *i->second);

    // this chunk is a bit elliptical for the sake of speed...
    if(r.Strength > worstbest) {
      r.TargetLinkPak = i->second;
      if(Best.size() == count) {
Best.pop();
Best.push(r);
worstbest = Best.top().Strength;
      }
      else {
Best.push(r);
if(Best.size() == count) worstbest = Best.top().Strength;
      }
    }
  }

  list<Relation> Nearest;
  while(Best.size() > 0) {
    Nearest.push_back(Best.top());
    Best.pop();
  }

  return Nearest;
}

// Returns a string representation of this archive
void LinkPakArchive::print(ostream& out)
{
  out << "! LinkPakArchive (" << Name << ")\n";

  for(ArcMap::iterator i = Archive.begin(); i != Archive.end(); i++)
    out << "! LinkPak " << i->second->id() << " (" << i->second->name()
<< ")\n";
}

list<LinkPak*> LinkPakArchive::Contents(void)
{
  list<LinkPak*> l;

  for(ArcMap::iterator i = Archive.begin(); i != Archive.end(); i++)
    l.push_back(i->second);

  return l;
}
```

LinkPakCompare.h

This header file contains object declarations for various LinkPakCompare objects, which are used to determine how similar LinkPaks are. They are described in detail in section 3.4.

```
#ifndef _LINK_PAK_COMPARE_H_
#define _LINK_PAK_COMPARE_H_

#include <vector>
#include <set>
#include "LinkPak.h"

// The default depth for the recursive compare functions below.
#define DEF_LPCR_DEPTH 3

/*
 * LinkPakCompare defines what in Java is called an "interface" for the
 * various classes that perform compares between LinkPaks. Hopefully this
 * approach won't cause a lack of inlining and hence slowness...
 */
class LinkPakCompare {
  public:
    virtual double Compare(LinkPak&, LinkPak&) = 0;
};

/*
 * LPCBase1 is a subclass of LinkPakCompare. It compares two LinkPaks
 * by summing up the differences in _only the Relations they have in
 * common_ and dividing by the number of Relations summed.
 */
class LPCBase1 : public LinkPakCompare {
  public:
    double Compare(LinkPak&, LinkPak&);
};

/*
 * LPCBase2 is a subclass of LinkPakCompare. It compares two LinkPaks
 * by summing up the differences in _only the Relations they have in
 * common_ and dividing by the total number of distinct Relations.
 */
class LPCBase2 : public LinkPakCompare {
  public:
    double Compare(LinkPak&, LinkPak&);
};

/*
 * LPCBase3 is a subclass of LinkPakCompare. It compares two LinkPaks
 * by summing up the differences between all common Relations, summing
 * up all uncommon Relations, then dividing by the total number of
 * distinct Relations.
 */
class LPCBase3 : public LinkPakCompare {
  public:
    double Compare(LinkPak&, LinkPak&);
};

/*
 * LPCBase4 is a subclass of LinkPakCompare. It functions in a conceptually
 * identical way to LPCBase3, except that it sets the difference between
 * Relations the two LinkPaks don't have in common to 1.
 */
class LPCBase4 : public LinkPakCompare {
  public:
    double Compare(LinkPak&, LinkPak&);
};

/*
 * LPCRecursive is a subclass of LinkPakCompare. It compares two LinkPaks
 * in a manner similar to Compare3. However, when it encounters a Relation
 * in one LinkPak missing from the other, it performs a recursive compare
 * on the lacking LinkPak and comes up with the needed value. In this
 * fashion a more accurate comparison is achieved.
```

```
 */
class LPCRecursive : public LinkPakCompare {
  private:
    int recdepth;
    LinkPakCompare* BaseCompare;
    double CompareRecurser(LinkPak&, LinkPak&, int);
  public:
    LPCRecursive(LinkPakCompare*, int=DEF_LPCR_DEPTH);
    double Compare(LinkPak&, LinkPak&);
};

/*
 * LPCRecursiveAdd is a subclass of LinkPakCompare. It does exactly what
 * LPCRecursive does but also adds the missing Relations to LinkPaks,
 * thus eliminating the need to recurse again later.
 */
class LPCRecursiveAdd : public LinkPakCompare {
  private:
    int recdepth;
    LinkPakCompare* BaseCompare;
    set<LinkPak*> OffLimits;
    double CompareRecurser(LinkPak&, LinkPak&, int);
  public:
    LPCRecursiveAdd(LinkPakCompare*, set<LinkPak*>&, int=DEF_LPCR_DEPTH);
    double Compare(LinkPak&, LinkPak&);
    virtual ~LPCRecursiveAdd() { } // why does the compiler want this?
};

/*
 * LPCInsert compares two LinkPaks with the LinkPakCompare object provided
 * to the constructor. When it's done, it adds a Relation to the first
 * LinkPak (Strength is set to the result of the search, naturally) to
 * the second LinkPak
 */
class LPCInsert : public LinkPakCompare {
  private:
    LinkPakCompare* C;
  public:
    LPCInsert(LinkPakCompare*);
    double Compare(LinkPak&, LinkPak&);
    virtual ~LPCInsert() { }
};

#endif
```

LinkPakCompare.cc

This code implements the several LinkPakCompare class, which are described in detail in section 3.4.

```
#include <math.h>
#include <vector>
#include <set>

#ifdef DEBUG
#include <iostream>
#endif

#include "LinkPak.h"
#include "LinkPakCompare.h"
#include "Relation.h"

#define UNLESS(item) if(!( (item) ))

/*
 * LPCBase1 is a subclass of LinkPakCompare. It compares two LinkPaks
 * by summing up the differences in _only the Relations they have in
 * common_ and dividing by the number of Relations summed.
 */
double LPCBase1::Compare(LinkPak& l, LinkPak& m)
{
  // be sure that they aren't the same LinkPaks...
  if(l == m) return 1.0;

  double difference = 0;
  int comparecount = 0;

  for(PakMap::iterator i = l.Pak.begin(); i != l.Pak.end(); i++)
    if(Relation *them = m.find(PAK_KEY_UNCAST (i->first))) {
      difference += fabs(i->second.Strength - them->Strength);
      comparecount++;
    }

#ifdef DEBUG
  cout << "* LPCBase1 made " << comparecount << " comparisons." << endl;
#endif

  // The 0 here is deliberate. If the two LinkPaks have nothing in common,
  // we assume for the purposes of this function that they're not much
  // alike at all.
  return comparecount == 0 ? 0 : 1 - (difference / comparecount);
}

/*
 * LPCBase2 is a subclass of LinkPakCompare. It compares two LinkPaks
 * by summing up the differences in _only the Relations they have in
 * common_ and dividing by the total number of distinct Relations.
 */
double LPCBase2::Compare(LinkPak& l, LinkPak& m)
{
  // be sure that they aren't the same LinkPaks...
  if(l == m) return 1.0;

  double difference = 0;
  int comparecount = 0;

  for(PakMap::iterator i = l.Pak.begin(); i != l.Pak.end(); i++)
    if(Relation *them = m.find(PAK_KEY_UNCAST (i->first))) {
      difference += fabs(i->second.Strength - them->Strength);
      comparecount++;
    }

#ifdef DEBUG
  cout << "* LPCBase2 made " << comparecount << " actual comparisons." << endl;
#endif

  return 1 - (difference / (l.Pak.size() + (m.Pak.size() - comparecount)));
}
```

```
/*
 * LPCBase3 is a subclass of LinkPakCompare. It compares two LinkPaks
 * by summing up the differences between all common Relations, summing
 * up all uncommon Relations, then dividing by the total number of
 * distinct Relations.
 */
double LPCBase3::Compare(LinkPak& l, LinkPak& m)
{
  // be sure that they aren't the same LinkPaks...
  if(l == m) return 1.0;

  double difference = 0;

  for(PakMap::iterator i = l.Pak.begin(); i != l.Pak.end(); i++) {
    Relation *them = m.find(PAK_KEY_UNCAST (i->first));
    difference += them ? fabs(i->second.Strength - them->Strength) :
 i->second.Strength;
  }

  int comparecount = 0;

  for(PakMap::iterator i = m.Pak.begin(); i != m.Pak.end(); i++)
    UNLESS(l.find(PAK_KEY_UNCAST (i->first))) {
      difference += i->second.Strength;
      comparecount++;
    }

#ifdef DEBUG
  cout << "* LPCBase3 made " << l.Pak.size() + comparecount << " comparisons."
       << endl;
#endif

  return 1 - difference / (l.Pak.size() + comparecount);
}

/*
 * LPCBase4 is a subclass of LinkPakCompare. It functions in a conceptually
 * identical way to LPCBase3, except that it sets the difference between
 * Relations the two LinkPaks don't have in common to 1.
 */
double LPCBase4::Compare(LinkPak& l, LinkPak& m)
{
  // be sure that they aren't the same LinkPaks...
  if(l == m) return 1.0;

  double difference = 0;

  for(PakMap::iterator i = l.Pak.begin(); i != l.Pak.end(); i++) {
    Relation *them = m.find(PAK_KEY_UNCAST (i->first));
    difference += them ? fabs(i->second.Strength - them->Strength) : 1;
  }

  int comparecount = 0;

  for(PakMap::iterator i = m.Pak.begin(); i != m.Pak.end(); i++)
    UNLESS(l.find(PAK_KEY_UNCAST (i->first))) {
      difference += 1;
      comparecount++;
    }

#ifdef DEBUG
  cout << "* LPCBase3 made " << l.Pak.size() + comparecount << " comparisons."
       << endl;
#endif

  return 1 - difference / (l.Pak.size() + comparecount);
}

/*
 * LPCRecursive is a subclass of LinkPakCompare. It compares two LinkPaks
 * in a manner similar to Compare3. However, when it encounters a Relation
 * in one LinkPak missing from the other, it performs a recursive compare
```

```
 * on the lacking LinkPak and comes up with the needed value. In this
 * fashion a more accurate comparison is achieved.
 */
// The constructor supplies LPCRecursive with a LinkPakCompare object to
// use when it has recursed deep enough. I suppose you could use another
// recursive compare, but why would you want to do that? Meanwhile, depth
// specifies the depth of the recursion.
LPCRecursive::LPCRecursive(LinkPakCompare* c, int depth)
{
  BaseCompare = c;
  recdepth = depth;
}

// This function just kicks off the hidden recurser function named
// CompareRecurser
double LPCRecursive::Compare(LinkPak& l, LinkPak& m)
{
  return CompareRecurser(l, m, recdepth);
}

// This function does the actual work
double LPCRecursive::CompareRecurser(LinkPak& l, LinkPak& m, int depth)
{
  // be sure that they aren't the same LinkPaks...
  if(l == m) return 1.0;
  // This line derives from a particularly insidious bug that caused recursive
  // compares to result in NaNs whenever they compared (somewhere deep within
  // the recursive tree) two LinkPaks with no Relations whatsoever. An && would
  // thus be the most intuitive choice for the test below; however, since a
  // LinkPak with no Relations is not liable to be very similar to anything,
  // we extract a small speedup by using || instead.
  if((l.Pak.size() == 0) || (m.Pak.size() == 0)) return 0;

  if(depth == 0) return BaseCompare->Compare(l, m); // The base case

  double difference = 0;

  for(PakMap::iterator i = l.Pak.begin(); i != l.Pak.end(); i++)
    if(Relation *them = m.find(PAK_KEY_UNCAST (i->first)))
      difference += fabs(i->second.Strength - them->Strength);
    else
      difference += CompareRecurser(m, *(PAK_KEY_UNCAST (i->first)), depth-1);

  int comparecount = 0;

  for(PakMap::iterator i = m.Pak.begin(); i != m.Pak.end(); i++)
    UNLESS(l.find(PAK_KEY_UNCAST (i->first))) {
      difference += CompareRecurser(l, *(PAK_KEY_UNCAST (i->first)), depth-1);
      comparecount++;
    }

#ifdef DEBUG
  cout << "* LPCRecursive presently divulges no debug information" << endl;
#endif

  return 1 - difference / (l.Pak.size() + comparecount);
}

/*
 * LPCRecursiveAdd is a subclass of LinkPakCompare. It does exactly what
 * LPCRecursive does but also adds the missing Relations to LinkPaks,
 * thus eliminating the need to recurse again later.
 */
// The constructor supplies LPCRecursiveAdd with a LinkPakCompare object to
// use when it has recursed deep enough. I suppose you could use another
// recursive compare, but why would you want to do that? Meanwhile, depth
// specifies the depth of the recursion.
LPCRecursiveAdd::LPCRecursiveAdd(LinkPakCompare* c, set<LinkPak*>& OL,
 int depth)
{
  BaseCompare = c;
  OffLimits = OL;
```

```
  recdepth = depth;
}

// This function just kicks off the hidden recurser function named
// CompareRecurser
double LPCRecursiveAdd::Compare(LinkPak& l, LinkPak& m)
{
  return CompareRecurser(l, m, recdepth);
}

// This function does the actual work
double LPCRecursiveAdd::CompareRecurser(LinkPak& l, LinkPak& m, int depth)
{
  // be sure that they aren't the same LinkPaks...
  if(l == m) return 1.0;

  if(depth == 0) return BaseCompare->Compare(l, m); // The base case

  // For safety's sake, we add the same test as in LPCRecursive
  if((l.Pak.size() == 0) || (m.Pak.size() == 0)) return 0;

  double difference = 0;
  vector<LinkPak*> ToAdd;

  for(PakMap::iterator i = l.Pak.begin(); i != l.Pak.end(); i++)
    if(Relation *them = m.find(PAK_KEY_UNCAST (i->first)))
      difference += fabs(i->second.Strength - them->Strength);
    else UNLESS(OffLimits.find(PAK_KEY_UNCAST (i->first)) == OffLimits.end())
      ToAdd.push_back(PAK_KEY_UNCAST (i->first));

  // now we compute and add the new Relations after the initial scan:
  for(vector<LinkPak*>::iterator i = ToAdd.begin(); i != ToAdd.end(); i++) {
    double score = CompareRecurser(m, **i, depth-1);
    m.insert(*i, score, 0);
    difference += score;
  }

  // now we do it for the other LinkPak
  int comparecount = 0;

  for(PakMap::iterator i = m.Pak.begin(); i != m.Pak.end(); i++)
    UNLESS(l.find(i->first) ||
   (OffLimits.find(PAK_KEY_UNCAST (i->first)) != OffLimits.end())) {
      double score = CompareRecurser(l, *(PAK_KEY_UNCAST (i->first)), depth-1);
      l.insert(PAK_KEY_UNCAST (i->first), score, 0);
      difference += score;
      comparecount++;
    }

#ifdef DEBUG
  cout << "* LPCRecursiveAdd presently divulges no debug information" << endl;
#endif

  return 1 - difference / (l.Pak.size() + comparecount);
}

/*
 * LPCInsert compares two LinkPaks with the LinkPakCompare object provided
 * to the constructor. When it's done, it adds a Relation to the first
 * LinkPak (Strength is set to the result of the search, naturally) to
 * the second LinkPak
 */
// The constructor
LPCInsert::LPCInsert(LinkPakCompare* c) : C(c) { }

// The comparison - nothing to it!
double LPCInsert::Compare(LinkPak& l, LinkPak& m)
{
  double result = C->Compare(l, m);
  m.insert(&l, result, 0);
  return result;
}
```

LinkPakAverager.h

A LinkPakAverager is a LinkPak with a twist–its Relations are not inserted (customarily–it is possible, but it makes no sense) but are instead the averaged relations from other LinkPaks. LinkPakAveragers are potentially useful for deriving new actuator outputs from stored experience.

```
/*
 * A LinkPakAverager is a LinkPak whose Relations are the average of several
 * LinkPaks submitted to it for averaging. The LRU mechanism for LinkPaks
 * (perhaps an ill-chosen name - Least Recently Modified would be better)
 * allows for several speedups, but these are not currently implemented.
 */
#ifndef _LINK_PAK_AVERAGER_H_
#define _LINK_PAK_AVERAGER_H_

#include <string>
#include <hash_map.h>
//#include <map>
#include <hash_set.h>
//#include <set>

#include "LinkPak.h"
#include "LinkPakArchive.h"

typedef hash_map<PAK_KEY_TYPE, unsigned int> LinkCountMap;
//typedef map<PAK_KEY_TYPE, unsigned int> LinkCountMap;
typedef hash_set<PAK_KEY_TYPE> CheckedSet;
//typedef set<PAK_KEY_TYPE> CheckedSet;

class LinkPakAverager : public LinkPak {
  private:
    LinkCountMap LinkCount;
    LinkPak* myOwnPak;
    LinkPakArchive* AverageList;
    LinkPakArchive* CanUse;
    bool UpToDate;
  public:
    LinkPakAverager(string, LinkPakArchive*); // The constructor
    ~LinkPakAverager(void); // The destructor
    bool insert(Relation&); // Insert Relation into LinkPak
    bool insert(LinkPak*, double, unsigned long); // similar
    bool insertLinkPak(LinkPak*); // Add LinkPak to average list
    bool removeLinkPak(LinkPak*); // Remove LinkPak from same
    void update(void); // Updates averages
    // the only guarantee we will make about the following function is that
    // it will only return false during times when the averages truly aren't
    // updated. if it returns true, the averages may or may not be updated.
    bool thinksItsReady(void);
};

#endif
```

LinkPakAverager.cc

This code implements the LinkPakAverager class, which is described on the previous page.

```
/*
 * A LinkPakAverager is a LinkPak whose Relations are the average of several
 * LinkPaks submitted to it for averaging. Note that the actual averaging does
 * not take place until update is called. Insertions into a LinkPakAverager
 * are placed into a private LinkPak and averaged in with all the other
 * inserted LinkPaks.
 */
#include <string>
#include <list>
#include <set>
#include <hash_map.h>
//#include <map>
#include <hash_set.h>
//#include <set>

#include "LinkPak.h"
#include "LinkPakArchive.h"
#include "LinkPakAverager.h"

// The constructor
LinkPakAverager::LinkPakAverager(string n, LinkPakArchive* c) : LinkPak(n)
{
  CanUse = c;
  myOwnPak = new LinkPak("LinkPakAverager LinkPak");
  AverageList = new LinkPakArchive("LinkPakAverager Averaging List");
  UpToDate = false;
  AverageList->insert(myOwnPak);
}

// The destructor
LinkPakAverager::~LinkPakAverager()
{
  delete AverageList;
  delete myOwnPak;
}

// Insert Relation into LinkPak
bool LinkPakAverager::insert(Relation& r)
{
  return myOwnPak->insert(r);
}

// similar
bool LinkPakAverager::insert(LinkPak* l, double s, unsigned long id)
{
  return myOwnPak->insert(l, s, id);
}

// Add LinkPak to average list
bool LinkPakAverager::insertLinkPak(LinkPak* l)
{
  if(AverageList->insert(l)) {
    UpToDate = false;
    return true;
  }
  return false;
}

bool LinkPakAverager::removeLinkPak(LinkPak* l)
{
  if(AverageList->remove(l->id())) {
    UpToDate = false;
    return true;
  }
  return false;
}

void LinkPakAverager::update(void)
```

```
{
  // First we need to dump out the Pak.
  Pak.erase(Pak.begin(), Pak.end());

  CheckedSet Checked;
  list<LinkPak*> l = AverageList->Contents();
  list<LinkPak*>::iterator i, k;
  list<Relation> c;
  list<Relation>::iterator j;

  unsigned int lsize = l.size(); // so we don't have to compute it later
  for(i = l.begin(); i != l.end(); i++) {
    c = (*i)->Contents();
    // SLOW
    for(j = c.begin(); j != c.end(); j++) {
      // avoid concepts not in the CanUse archive
      if(!CanUse->find(j->TargetLinkPak->id())) continue;
      // avoid already visited concepts
      if(Checked.find(PAK_KEY_CAST j->TargetLinkPak) != Checked.end()) continue;
      Checked.insert(PAK_KEY_CAST j->TargetLinkPak);
      double avg = 0;
      for(k = i; k != l.end(); k++)
if(Relation* q = (*k)->find(j->TargetLinkPak)) avg += q->Strength;
      avg /= (myOwnPak->size() > 0) || (lsize == 1) ? lsize : lsize - 1;
      LinkPak::insert(j->TargetLinkPak, avg, 0);
    }
  }

  LRU = LRUctr++;
  UpToDate = true;
}

bool LinkPakAverager::thinksItsReady(void)
{
  if(!UpToDate) return false;

  list<LinkPak*> l = AverageList->Contents();
  list<LinkPak*>::iterator i;
  for(i = l.begin(); i != l.end(); i++)
    if((*i)->lru() > LRU) return false;

  return true;
}
```

LinkPakExtrapolator.h

A LinkPakExtrapolator is a LinkPak with a twist–its Relations are not inserted (customarily–it is possible, but it makes no sense) but are instead extrapolated from relations in other LinkPaks. Unlike LinkPakAveragers, LinkPakExtrapolators must be told exactly which relations to extrapolate. LinkPakExtrapolators are potentially useful for deriving new actuator outputs from stored experience. See also section 3.3.

```
/*
 * A LinkPakExtrapolator is a LinkPak whose Relations are either supplied or
 * extrapolated from near matches to supplied Relations. The extrapolation used
 * is linear least squared.
 */
#ifndef _LINK_PAK_EXTRAPOLATOR_H_
#define _LINK_PAK_EXTRAPOLATOR_H_

#include <string>
#include <list>

#include "LinkPak.h"
#include "LinkPakArchive.h"

class LinkPakExtrapolator {
  private:
    LinkPak* myOwnPak;
    LinkPakArchive* Archive;
    LinkPakCompare* Compare;
    unsigned int count;
  public:
    LinkPakExtrapolator(LinkPak*, LinkPakArchive*, LinkPakCompare*,
unsigned int); // The constructor
    LinkPak* getLinkPak(void); // returns addr of myLinkPak
    bool extrapolateAndInsert(list<LinkPak*>); // Sets up the extrapolation
    bool extrapolateAndInsertForever(list<LinkPak*>); // Sets up extrapolation
    list<Relation> extrapolateRelations(list<LinkPak*>); // The extrapolator
};

#endif
```

LinkPakExtrapolator.cc

This code implements the LinkPakExtrapolator class, which is described on the previous page.

```
/*
 * A LinkPakExtrapolator is a LinkPak whose Relations are either supplied or
 * extrapolated from near matches to supplied Relations. The extrapolation used
 * is linear least squared.
 */

#include <iostream>
#include <cmath>
#include <cstdio>
#include <string>
#include <list>

#include "Relation.h"
#include "LinkPak.h"
#include "LinkPakArchive.h"
#include "LinkPakExtrapolator.h"

//#define DEBUG_PLOT 1

// The constructor
LinkPakExtrapolator::LinkPakExtrapolator(LinkPak* l, LinkPakArchive* a,
 LinkPakCompare* C, unsigned int c)
  : myOwnPak(l), Archive(a), Compare(C), count(c) { }

// returns address of myOwnPak
LinkPak* LinkPakExtrapolator::getLinkPak(void) { return myOwnPak; }

// A wrapper function for the extrapolation
bool LinkPakExtrapolator::extrapolateAndInsert(list<LinkPak*> l)
{
  if(myOwnPak->size() == 0) return false;
  bool success = true;

  list<Relation> r = extrapolateRelations(l);
  for(list<Relation>::iterator i = r.begin(); i != r.end(); i++)
    if(!myOwnPak->insert(*i)) success = false;

  return success;
}

// The same, but with insertForever
bool LinkPakExtrapolator::extrapolateAndInsertForever(list<LinkPak*> l)
{
  if(myOwnPak->size() == 0) return false;
  bool success = true;

  list<Relation> r = extrapolateRelations(l);
  for(list<Relation>::iterator i = r.begin(); i != r.end(); i++)
    if(!myOwnPak->insertForever(*i)) success = false;

  return success;
}

// The actual extrapolator.
//
// An earlier extrapolator was created that tried to find a linear function
// from the known elements of myOwnPak to those listed in l using least-squared
// analysis methods and matrix solving. Unfortunately, this technique resulted
// in the production of many singular matricies, crashing the program. There
// may be a more intelligent way to solve matricies than Gaussian elimination,
// but the math is beyond me and I don't want to use Numerical Recipes. In
// any case, the present method is employed, though it may seem a bit ad-hoc.
// It is:
//  1. Find the n nearest matches to myOwnPak.
//  2. Find a linear function from the strength of the matches to the desired
//     element in l.
//  3. Compute the value for strength=1 (for naturally we match ourselves
//     perfectly) and insert into myOwnPak.
```

```
//  4. Repeat for all elements of l.
//
// This code adapted from code found at
//   http://astronomy.swin.edu.au/pbourke/,
// Thanks Paul for your great pages!
list<Relation> LinkPakExtrapolator::extrapolateRelations(list<LinkPak*> l)
{
  list<Relation> result;
  LinkPak tmpPak = *myOwnPak;

  for(list<LinkPak*>::iterator i = l.begin(); i != l.end(); ++i) {
    list<Relation> sims = Archive->FindNearestNTo(count, &tmpPak, *Compare);
    int numvars = sims.size();
    Relation r;

    r.TargetLinkPak = *i;
    r.id = 0;

/*
cout << "Dumping sims:" << endl;
for(list<Relation>::iterator q = sims.begin(); q != sims.end(); ++q)
  cout << q->TargetLinkPak->id() << " (" << q->TargetLinkPak->name() <<
") at " << q->Strength << endl;
{ bool FAULT=false;
  for(list<Relation>::iterator x = sims.begin(); x != sims.end(); ++x)
    if(x->Strength == 1) {
      FAULT=true;
      cout << "FAULT!" << endl;
      x->TargetLinkPak->print(cout);
    }
  if(FAULT == true) {
    tmpPak.print(cout);
    exit(-1);
  } }
*/
#ifdef DEBUG_PLOT
cout << "Creating sample plot:" << endl;
char Plot[23][63];
for(int j = 0; j < 23; ++j)
  for(int k = 0; k < 63; ++k)
    if(k == 62) Plot[j][k] = '\0';
    else if((j == 21) && (k == 1)) Plot[j][k] = '+';
    else if((j == 2) && (k == 0)) Plot[j][k] = '1';
    else if((j == 20) && (k == 0)) Plot[j][k] = '0';
    else if(k == 1) Plot[j][k] = '|';
    else if(j == 21) Plot[j][k] = '-';
    else Plot[j][k] = ' ';

double minval = sims.begin()->Strength;
{ char tmpnum[15];
  sprintf(tmpnum, "%f", minval);
  char *p = &Plot[22][3];
  for(char *c = tmpnum; *c; *p++ = *c++); }

double maxval;
{ list<Relation>::iterator q = sims.end(); q--;
  maxval = q->Strength;
  char tmpnum[15];
  sprintf(tmpnum, "%f", maxval);
  char *p = &Plot[22][50];
  for(char *c = tmpnum; *c; *p++ = *c++); }

double gap = maxval - minval;
for(list<Relation>::iterator q = sims.begin(); q != sims.end(); ++q) {
  int sx;
  if(gap > 0) sx = (int) ((47.0 * ((q->Strength - minval) / gap)) + 3.0);
  else sx = 50;
  int sy;
  if(Relation *y = q->TargetLinkPak->find(*i))
    sy = (int) (20.0 - (18.0 * y->Strength));
  else sy = 20;
```

```
  Plot[sy][sx] = '+';
}
#endif

    double sxx, sumx=0;

    { double sumx2=0;
      for(list<Relation>::iterator x = sims.begin(); x != sims.end(); ++x) {
sumx += x->Strength;
sumx2 += x->Strength * x->Strength;
      }
      sxx = sumx2 - sumx * sumx / numvars; }

    if(sxx != 0.0) {
      double sxy, sumy=0;

      { double sumxy=0;
for(list<Relation>::iterator x = sims.begin(); x != sims.end(); ++x)
  if(Relation* y = x->TargetLinkPak->find(*i)) {
    sumy += y->Strength;
    sumxy += x->Strength * y->Strength;
  }
  else {
    cout << "\a RED FLAG: couldn't find this node!" << endl;
    (*i)->print(cout);
  }
sxy = sumxy - sumx * sumy / numvars; }

      double a = sxy / sxx;

      r.Strength = a + (sumy - a * sumx) / numvars;
      if(r.Strength < 0) r.Strength = 0;
      else if(r.Strength > 1) r.Strength = 1;
#ifdef DEBUG_PLOT
double gapjump = a * gap;
char plotchar;
if(fabs(gapjump) < 0.7) plotchar = '-';
else if(fabs(gapjump) > 3) plotchar = '|';
else if(gapjump > 0) plotchar = '/';
else plotchar = '\\';

double interval = (maxval - minval) / 48.0;
double b = (sumy - (a * sumx)) / numvars;
double ix = minval;
for(int cx = 3; cx <= 50; ++cx) {
  int cy = (int) (((a * ix) + b) * 18.0);
  if((cy > 18) || (cy < 0)) continue;
  cy = 20 - cy;
  if(Plot[cy][cx] == ' ') Plot[cy][cx] = plotchar;
  ix += interval;
}
#endif
    }
    // This special case occurs if all of our samples have the exact same
    // Strength value. Since there is no line defined for this case, we'll
    // just average all of the values of our samples and use that as the
    // strength value.
    else {
      r.Strength = 0;

cout << '!';

      for(list<Relation>::iterator j = sims.begin(); j != sims.end(); ++j)
if(Relation* y = j->TargetLinkPak->find(*i)) r.Strength += y->Strength;

      r.Strength /= numvars;
    }

#ifdef DEBUG_PLOT
for(int y=0; y<23; ++y) cout << '\t' << Plot[y] << endl;
#endif
```

```
    result.push_back(r);
    tmpPak.insertForever(r);
  }

  return result;
}
```

Interface.h

The first comment in this file underscores its chief merits. The Interface class and its LinkPakModifier helper class (both defined here) make clever use of object oriented design to render libdydex programs simple to make. Here's how it works:

A LinkPakModifier is a class that does things to LinkPaks–adds relations to them, inserts them into a LinkPakArchive, and so on. The Interface class, in addition to various other nice things, contains two strings of LinkPakModifiers: one for the training phase, and one for the execution phase. When the TrainOnce() method of an Interface object is called, each LinkPakModifier in the training phase string is applied once to the LinkPak representing the current node. Likewise, when the ActOnce() method is called, the same thing happens with the execution phase string. Modifying the exact training method of LinkPaks, choosing whether state should be stored, and many other consequential choices can be affected simply by rearranging the LinkPakModifier objects in these strings.

```
// We are object oriented out the proverbial WAZOO.
#ifndef _INTERFACE_H_
#define _INTERFACE_H_

#include <iostream>
#include <set>
#include <list>

#include "LinkPak.h"
#include "LinkPakArchive.h"
#include "LinkPakCompare.h"
#include "LinkPakAverager.h"
#include "LimitedLinkPak.h"

typedef set<LinkPak*> BasicConcepts;

// Predefinitions
class NewPakMaker;
class LinkPakModifier;

class Interface {
  friend class LinkPakModifier;
  private:
    NewPakMaker *PM;
    list<LinkPakModifier*> TrainMs, ActMs;
    LinkPakArchive* Archive;
    BasicConcepts Inputs, Outputs;
  public:
    Interface();
    ~Interface();

    LinkPak* RegisterInput(string);
    LinkPak* RegisterOutput(string);
    void RegisterNewPakMaker(NewPakMaker*);
    void RegisterTrainingModifiers(list<LinkPakModifier*>);
    void RegisterActionModifiers(list<LinkPakModifier*>);

    void TrainOnce();
    void ActOnce();
};

// we'll just put these NewPakMaker classes in the .h file - they're so simple
class NewPakMaker {
  public:
    virtual LinkPak* operator()(void)
      { return new LinkPak("NewPakMaker LinkPak"); }
};

class NewLLPMaker : public NewPakMaker {
  private:
    unsigned long size;
  public:
    NewLLPMaker(unsigned long s) : size(s) { }
    virtual LinkPak* operator()(void) {
      return new LimitedLinkPak("NewPakMaker LimitedLinkPak", size);
    }
};
```

```
// the LinkPakModifier classes, on the other hand, must be elaborated in the
// .cc file. Is there a better way to pass all this info? We can't befriend
// every single LinkPakModifier subclass...
class LinkPakModifier {
  private:
    Interface *I;
  protected:
    LinkPakArchive* getArchive();
    BasicConcepts* getInputs();
    BasicConcepts* getOutputs();
  public:
    LinkPakModifier(Interface*);
    virtual void operator()(LinkPak*) = 0;
};

// These are pretty standard LinkPakModifier classes that will probably
// see some use in many programs
class LPMAddInputs : public LinkPakModifier {
  private:
    list<Relation> myInputs;
  public:
    LPMAddInputs(Interface*);
    void reset(void);
    bool AddInput(Relation r);
    void operator()(LinkPak*);

    virtual ~LPMAddInputs() { }
};

class LPMAddOutputs : public LinkPakModifier {
  private:
    list<Relation> myOutputs;
  public:
    LPMAddOutputs(Interface*);
    void reset(void);
    bool AddOutput(Relation r);
    void operator()(LinkPak*);

    virtual ~LPMAddOutputs() { }
};

class LPMAddSimilarities : public LinkPakModifier {
  private:
    unsigned int count;
    LinkPakCompare *Compare;
  public:
    LPMAddSimilarities(Interface*, unsigned int, LinkPakCompare*);
    void operator()(LinkPak*);
};

class LPMAddSimilarities2 : public LinkPakModifier {
  private:
    unsigned int count;
    LinkPakCompare *Compare;
  public:
    LPMAddSimilarities2(Interface*, unsigned int, LinkPakCompare*);
    void operator()(LinkPak*);
};

class LPMDoAverageHistory : public LinkPakModifier {
  private:
    unsigned int count;
    list<LinkPak*> History;
    LinkPakAverager* HistAverager;
  public:
    LPMDoAverageHistory(Interface*, unsigned int);
    void reset(void);
    void operator()(LinkPak*);

    virtual ~LPMDoAverageHistory() { }
};
```

```
class LPMDeriveAddOutputs : public LinkPakModifier {
  private:
    unsigned int count;
    LinkPakCompare *Compare;
    list<Relation> Results;
  public:
    list<Relation>& getResults(void);
    LPMDeriveAddOutputs(Interface*, unsigned int, LinkPakCompare*);
    void operator()(LinkPak*);

    virtual ~LPMDeriveAddOutputs() { }
};

class LPMExtrapolateOutputs : public LinkPakModifier {
  private:
    unsigned int count;
    LinkPakCompare *Compare;
    list<Relation> Results;
  public:
    list<Relation>& getResults(void);
    LPMExtrapolateOutputs(Interface*, unsigned int, LinkPakCompare*);
    void operator()(LinkPak*);

    virtual ~LPMExtrapolateOutputs() { }
};

// This LinkPakModifier doesn't really modify per se
class LPMPrint : public LinkPakModifier {
  private:
    ostream& out;
  public:
    LPMPrint(Interface*, ostream&);
    void operator()(LinkPak*);

    virtual ~LPMPrint() { }
};

// Neither does this one
class LPMPrintOutputs : public LinkPakModifier {
  private:
    ostream& out;
  public:
    LPMPrintOutputs(Interface*, ostream&);
    void operator()(LinkPak*);

    virtual ~LPMPrintOutputs() { }
};

// Neither does this one
class LPMChooseToInsert : public LinkPakModifier {
  private:
    double threshold;
    LinkPakCompare *Compare;
  public:
    LPMChooseToInsert(Interface*, double, LinkPakCompare*);
    void operator()(LinkPak*);
};

// Neither does this one
class LPMUsleep : public LinkPakModifier {
  private:
    long howlong;
  public:
    LPMUsleep(Interface*, long);
    void operator()(LinkPak*);
};

#endif
```

Interface.cc

This code implements the Interface and several LinkPakModifier classes, which are described on a prior page.

```
#include <set>
#include <list>

#include <sys/time.h>  // this is for Usleep
#include <sys/times.h> // so is this
#include <unistd.h>    // so is this

#include "Interface.h"
#include "LinkPak.h"
#include "LinkPakArchive.h"
#include "LinkPakCompare.h"
#include "LinkPakAverager.h"
#include "LinkPakExtrapolator.h"
#include "LimitedLinkPak.h"

Interface::Interface()
{
  Archive = new LinkPakArchive("Interface Archive");
}

Interface::~Interface()
{
  delete Archive;
}

LinkPak* Interface::RegisterInput(string s)
{
  LinkPak* l = new LinkPak(s);
  Inputs.insert(l);
  return l;
}

LinkPak* Interface::RegisterOutput(string s)
{
  LinkPak* l = new LinkPak(s);
  Outputs.insert(l);
  return l;
}

void Interface::RegisterNewPakMaker(NewPakMaker* NPM)
{
  PM = NPM;
}

void Interface::RegisterTrainingModifiers(list<LinkPakModifier*> ms)
{
  TrainMs = ms;
}

void Interface::RegisterActionModifiers(list<LinkPakModifier*> ms)
{
  ActMs = ms;
}

// wow, how easy is this?!
void Interface::TrainOnce()
{
  LinkPak* NewPak = (*PM)();
  list<LinkPakModifier*>::iterator i;

  for(i = TrainMs.begin(); i != TrainMs.end(); i++) (**i)(NewPak);
}

void Interface::ActOnce()
{
  LinkPak* NewPak = (*PM)();
  list<LinkPakModifier*>::iterator i;

  for(i = ActMs.begin(); i != ActMs.end(); i++) (**i)(NewPak);
```

```
}

LinkPakModifier::LinkPakModifier(Interface* i) : I(i) { }
LinkPakArchive* LinkPakModifier::getArchive(void) { return I->Archive; }
BasicConcepts* LinkPakModifier::getInputs(void) { return &(I->Inputs); }
BasicConcepts* LinkPakModifier::getOutputs(void) { return &(I->Outputs); }

LPMAddInputs::LPMAddInputs(Interface* i) : LinkPakModifier(i) { }

void LPMAddInputs::reset()
{
  myInputs.erase(myInputs.begin(), myInputs.end());
}

bool LPMAddInputs::AddInput(Relation r)
{
  BasicConcepts* Inputs = getInputs();

  if(Inputs->find(r.TargetLinkPak) != Inputs->end()) {
    myInputs.push_back(r);
    return true;
  }
  return false;
}

void LPMAddInputs::operator()(LinkPak* l)
{
  for(list<Relation>::iterator i = myInputs.begin(); i != myInputs.end(); i++)
    l->insertForever(*i);
}

LPMAddOutputs::LPMAddOutputs(Interface* i) : LinkPakModifier(i) { }

void LPMAddOutputs::reset()
{
  myOutputs.erase(myOutputs.begin(), myOutputs.end());
}

bool LPMAddOutputs::AddOutput(Relation r) {
  BasicConcepts* Outputs = getOutputs();

  if(Outputs->find(r.TargetLinkPak) != Outputs->end()) {
    myOutputs.push_back(r);
    return true;
  }
  return false;
}

void LPMAddOutputs::operator()(LinkPak* l)
{
  for(list<Relation>::iterator i = myOutputs.begin(); i != myOutputs.end(); i++)
    l->insertForever(*i);
}

LPMAddSimilarities::LPMAddSimilarities(Interface* i, unsigned int c,
LinkPakCompare* comp)
  : LinkPakModifier(i), count(c), Compare(comp) { }

void LPMAddSimilarities::operator()(LinkPak* l)
{
  if(count == 0) return;

  list<Relation> sims = getArchive()->FindNearestNTo(count, l, *Compare);
  for(list<Relation>::iterator i = sims.begin(); i != sims.end(); i++)
    l->insert(*i);
}

LPMAddSimilarities2::LPMAddSimilarities2(Interface* i, unsigned int c,
 LinkPakCompare* comp)
  : LinkPakModifier(i), count(c), Compare(comp) { }

void LPMAddSimilarities2::operator()(LinkPak* l)
```

```
{
  if(count == 0) return;

  BasicConcepts* Inputs = getInputs();
  BasicConcepts* Outputs = getOutputs();

  Relation r;
  r.id = 0;
  list<Relation>::iterator i, j;

  list<Relation> sims = getArchive()->FindNearestNTo(count, l, *Compare);
  for(i = sims.begin(); i != sims.end(); i++) {
    list<Relation> c = i->TargetLinkPak->Contents();
    for(j = c.begin(); j != c.end(); j++) {
      if(Inputs->find(j->TargetLinkPak) != Inputs->end()) continue;
      if(Outputs->find(j->TargetLinkPak) != Outputs->end()) continue;
      r.TargetLinkPak = j->TargetLinkPak;
      r.Strength = i->Strength * j->Strength;
      l->insert(r);
    }
  }
}

LPMDoAverageHistory::LPMDoAverageHistory(Interface* i, unsigned int c)
  : LinkPakModifier(i), count(c)
{
  HistAverager = new LinkPakAverager("LPMDoAverageHistory Averager",
     getArchive());
}

void LPMDoAverageHistory::reset()
{
  while(History.size() > 0) {
    HistAverager->removeLinkPak(History.back());
    History.pop_back();
  }
}

void LPMDoAverageHistory::operator()(LinkPak* l)
{
  if(count == 0) return;

  HistAverager->update();
  list<Relation> HAContents = HistAverager->Contents();
  for(list<Relation>::iterator i = HAContents.begin();
      i != HAContents.end();
      i++) l->insert(*i);
  History.push_front(l);
  HistAverager->insertLinkPak(l);
  while(History.size() > count) {
    HistAverager->removeLinkPak(History.back());
    History.pop_back();
  }
}

LPMDeriveAddOutputs::LPMDeriveAddOutputs(Interface* i, unsigned int c,
LinkPakCompare* comp)
  : LinkPakModifier(i), count(c), Compare(comp) { }

void LPMDeriveAddOutputs::operator()(LinkPak* l)
{
  if(count == 0) return;

  Results.erase(Results.begin(), Results.end());

  list<Relation> sims = getArchive()->FindNearestNTo(count, l, *Compare);
  set<LinkPak*> Checked;

  BasicConcepts* Outputs = getOutputs();

  list<Relation> c;
  list<Relation>::iterator i, j, k;
```

```
  Relation result;
  result.id = 0;

  for(i = sims.begin(); i != sims.end(); i++) {
    c = i->TargetLinkPak->Contents();

    for(j = c.begin(); j != c.end(); j++) {
      if(Outputs->find(j->TargetLinkPak) == Outputs->end()) continue;
      if(Checked.find(j->TargetLinkPak) != Checked.end()) continue;
      Checked.insert(j->TargetLinkPak);

      double avg = 0;
      for(k = i; k != sims.end(); k++)
if(Relation* r = k->TargetLinkPak->find(j->TargetLinkPak))
  // This might be handy also... later
  //avg += k->Strength * r->Strength;
  avg += r->Strength;

      avg /= count;
      result.TargetLinkPak = j->TargetLinkPak;
      result.Strength = avg;
      l->insertForever(result);
      Results.push_back(result);
    }
  }
}

list<Relation>& LPMDeriveAddOutputs::getResults()
{
  return Results;
}

LPMExtrapolateOutputs::LPMExtrapolateOutputs(Interface* i, unsigned int c,
LinkPakCompare* comp)
  : LinkPakModifier(i), count(c), Compare(comp) { }

void LPMExtrapolateOutputs::operator()(LinkPak* l)
{
  LinkPakExtrapolator LPE(l, getArchive(), Compare, count);

  list<LinkPak*> OutList;
  BasicConcepts* Outs = getOutputs();
  for(BasicConcepts::iterator i = Outs->begin(); i != Outs->end(); i++)
    OutList.push_back(*i);

  //LPE.extrapolateAndInsertForever(OutList);
  Results = LPE.extrapolateRelations(OutList);
  for(list<Relation>::iterator i = Results.begin(); i != Results.end(); i++)
    l->insertForever(*i);
}

list<Relation>& LPMExtrapolateOutputs::getResults()
{
  return Results;
}

LPMPrint::LPMPrint(Interface* i, ostream& o) : LinkPakModifier(i), out(o) { }

void LPMPrint::operator()(LinkPak* l)
{
  l->print(out);
}

LPMPrintOutputs::LPMPrintOutputs(Interface* i, ostream& o)
  : LinkPakModifier(i), out(o) { }

void LPMPrintOutputs::operator()(LinkPak* l)
{
  BasicConcepts* Outputs = getOutputs();
  list<Relation> c = l->Contents();

  for(list<Relation>::iterator i = c.begin(); i != c.end(); i++)
```

```
    if(Outputs->find(i->TargetLinkPak) != Outputs->end())
      out << "*** " << i->TargetLinkPak->id() << " ("
  << i->TargetLinkPak->name() << ")\t- " << i->Strength << endl;
}

LPMChooseToInsert::LPMChooseToInsert(Interface* i, double t,
LinkPakCompare* c)
  : LinkPakModifier(i), Compare(c)
{
  threshold = 1 - t;
}

// There are memory conservation issues with this not returning anything.
// For now, it just deletes the LinkPak. Thus, this object must be last on
// the execution chain.
void LPMChooseToInsert::operator()(LinkPak* l)
{
  LinkPakArchive* Archive = getArchive();

  if((threshold == 1) || (Compare == NULL) || (Archive->size() == 0))
    Archive->insert(l);
  else {
    list<Relation> sims = Archive->FindNearestNTo(1, l, *Compare);
    if(sims.front().Strength > threshold) delete l;
    else Archive->insert(l);
  }
}

LPMUsleep::LPMUsleep(Interface *i, long h)
  : LinkPakModifier(i), howlong(h) { }

void LPMUsleep::operator()(LinkPak* l)
{
  // This is an ugly cheat, but it works okay.
  struct timeval t;

  t.tv_sec = 0;
  t.tv_usec = howlong;

  select(0, NULL, NULL, NULL, &t);
}
```

Robot.cc

This code uses the libdydex library and Saphira to control the AmigoBOT robot. A full third of it is concerned with setting up Saphira, and a sizeable portion of this third is involved in turning off the bulk of Saphira's higher-level functionality. Of particular interest is the function SetupInterface, which prepares the lists of LinkPakModifiers for this program's Interface object. Note further that with this preparation out of the way, the code involved in training (Train() function) and execution (Act() function) is slim indeed.

```
#include <cmath>
#include <iostream>
#include <string>
#include <vector>
#include <list>

#include "Interface.h"
#include "LinkPak.h"
#include "Relation.h"

#ifndef SAPHIRA
#error Please define the SAPHIRA variable in the Makefile
#endif

#define MAX_VEL 400
#define VEL_INCR 40
#define MAX_RVEL 400
#define RVEL_INCR 40

#define SOLARIS

extern "C" {
#include "saphira.h"
}

vector<LinkPak*> Sonars(8), Bumpers(2);
LinkPak* VelIn;
LinkPak* VelOut;
LinkPak* RotVelIn;
LinkPak* RotVelOut;
int DesiredVel = 0, DesiredRotVel = 0;

bool OnMyOwn = false;
bool CanMove = true;

Interface I;
LPMAddInputs AI(&I);
LPMAddOutputs AO(&I);
LPMDoAverageHistory* DAH;
//LPMExtrapolateOutputs* DAO;
LPMDeriveAddOutputs* DAO;

// The compare object of choice
// LPCRecursive Compare(new LPCBase3, 1);
LPCBase3 Compare;

extern char **environ;

void SetupEnviron(void)
{
  int envsize = 0;

  for(char** erunner = environ; *erunner; erunner++)
    if(!strncmp("SAPHIRA=", *erunner, 8)) return;
    else envsize++;

  char** newenv = new char*[envsize+2];
  newenv[envsize+1] = NULL;
  newenv[envsize] = new char[8 + strlen(SAPHIRA) + 1];
  strcpy(newenv[envsize], "SAPHIRA=");
  strcat(newenv[envsize], SAPHIRA);

  char** er = environ;
  char** ner = newenv;
```

```
  while(*er) {
    *ner++ = *er++;
  }

  environ = newenv;
}

void SetupInterface(char **argv)
{
  DAH = new LPMDoAverageHistory(&I, atoi(argv[3]));
  //DAO = new LPMExtrapolateOutputs(&I, atoi(argv[4]), &Compare);
  DAO = new LPMDeriveAddOutputs(&I, atoi(argv[4]), &Compare);

  list<LinkPakModifier*> TMS, AMS;

  TMS.push_back(&AI);
  TMS.push_back(&AO);
  //TMS.push_back(DAH);
  TMS.push_back(new LPMAddSimilarities(&I, 1, &Compare));
  TMS.push_back(new LPMAddSimilarities2(&I, atoi(argv[2]), &Compare));
  //TMS.push_back(new LPMUsleep(&I, 500000));
  TMS.push_back(new LPMChooseToInsert(&I, atof(argv[5]), &Compare));

  AMS.push_back(&AI);
  //AMS.push_back(DAH);
  AMS.push_back(new LPMAddSimilarities2(&I, atoi(argv[2]), &Compare));
  //AMS.push_back(new LPMUsleep(&I, 500000));
  AMS.push_back(DAO);
  //AMS.push_back(new LPMPrintOutputs(&I, cout));
  //AMS.push_back(new LPMChooseToInsert(&I, atof(argv[5]), &Compare));

  I.RegisterNewPakMaker(new NewLLPMaker(atoi(argv[1])));
  I.RegisterTrainingModifiers(TMS);
  I.RegisterActionModifiers(AMS);
}

void SetupBasicConcepts(void)
{
  Sonars[0] = I.RegisterInput("sonar 0");
  Sonars[1] = I.RegisterInput("sonar 1");
  Sonars[2] = I.RegisterInput("sonar 2");
  Sonars[3] = I.RegisterInput("sonar 3");
  Sonars[4] = I.RegisterInput("sonar 4");
  Sonars[5] = I.RegisterInput("sonar 5");
  Sonars[6] = I.RegisterInput("sonar 6");
  Sonars[7] = I.RegisterInput("sonar 7");

  VelIn = I.RegisterInput("forward velocity input");
  RotVelIn = I.RegisterInput("rotational velocity input");

  Bumpers[0] = I.RegisterInput("left motor stall");
  Bumpers[1] = I.RegisterInput("right motor stall");

  VelOut = I.RegisterOutput("forward velocity output");
  RotVelOut = I.RegisterOutput("rotational velocity output");
}

void SetInputs(void)
{
  Relation r;
  r.id = 0;

  AI.reset();

  // Sonars
  for(int i = 0; i < 8; i++) {
    r.TargetLinkPak = Sonars[i];
// perhaps make this exponential instead of linear?
// is 5000 the right number?
    r.Strength = 1 - (sfSonarRange(i) / (double) 5000.0);
// just in case:
    if(r.Strength < 0) r.Strength = 0;
```

```
    else if(r.Strength > 1) r.Strength = 1;
    AI.AddInput(r);
  }

  // Wheels
  r.TargetLinkPak = VelIn;
  r.Strength = ((double) sfRobot.tv / (2 * MAX_VEL)) + 0.5;
// just in case:
  if(r.Strength < 0) r.Strength = 0;
  else if(r.Strength > 1) r.Strength = 1;
  AO.AddOutput(r);
  r.TargetLinkPak = RotVelIn;
  r.Strength = ((double) sfRobot.rv / (2 * MAX_RVEL)) + 0.5;
// just in case:
  if(r.Strength < 0) r.Strength = 0;
  else if(r.Strength > 1) r.Strength = 1;
  AO.AddOutput(r);

  // Bumpers
  r.TargetLinkPak = Bumpers[0];
  r.Strength = sfStalledMotor(sfLEFT);
  AI.AddInput(r);
  r.TargetLinkPak = Bumpers[1];
  r.Strength = sfStalledMotor(sfRIGHT);
  AI.AddInput(r);
}

void SetOutputs(void)
{
  Relation r;
  r.id = 0;

  AO.reset();

  // Wheels
  r.TargetLinkPak = VelOut;
  r.Strength = ((double) DesiredVel / (2 * MAX_VEL)) + 0.5;
// just in case:
  if(r.Strength < 0) r.Strength = 0;
  else if(r.Strength > 1) r.Strength = 1;
  AO.AddOutput(r);
  r.TargetLinkPak = RotVelOut;
  r.Strength = ((double) DesiredRotVel / (2 * MAX_VEL)) + 0.5;
// just in case:
  if(r.Strength < 0) r.Strength = 0;
  else if(r.Strength > 1) r.Strength = 1;
  AO.AddOutput(r);
}

// So simple!
void Train(void)
{
  SetInputs();
  SetOutputs();
  I.TrainOnce();
}

// OK, act is more complex
void Act(void)
{
  SetInputs();
  I.ActOnce();
  list<Relation> L(DAO->getResults());

  for(list<Relation>::iterator i = L.begin(); i != L.end(); i++)
    if(CanMove)
      if(i->TargetLinkPak == VelOut)
sfSetVelocity(DesiredVel = (int)((i->Strength - 0.5) * 2 * MAX_VEL));
      else if(i->TargetLinkPak == RotVelOut)
sfSetRVelocity(DesiredRotVel = (int)((i->Strength - 0.5)*2*MAX_RVEL));
}
```

```
extern "C" int myKeyFn(int ch)
{
  switch(ch) {
    case ' ':
sfMessage("         EMERGENCY STOP!     EMERGENCY STOP!");
  sfSetVelocity(DesiredVel = 0);
sfSetRVelocity(DesiredRotVel = 0);
if(OnMyOwn) CanMove = false;
return 1;
    case '!':
if(OnMyOwn) {
  sfMessage("Returning to training mode and user control.");
  OnMyOwn = false;
  cout << endl << endl << "t:";
}
else {
  sfMessage("Entering Action mode. Robot is not controllable.");
  sfMessage("       STRIKE SPACE BAR FOR EMERGENCY STOP");
  OnMyOwn = true;
  CanMove = true;
  cout << endl << endl << "A!:";
}
return 1;
    case 'G':
    case 'g':
if(OnMyOwn) {
  CanMove = true;
  return 1;
}
return 0;
    case 'I':
    case 'i':
if(OnMyOwn) return 0;
if(DesiredVel < MAX_VEL) sfSetVelocity(DesiredVel += VEL_INCR);
return 1;
    case 'J':
    case 'j':
if(OnMyOwn) return 0;
if(DesiredRotVel < MAX_RVEL) sfSetRVelocity(DesiredRotVel += RVEL_INCR);
return 1;
    case 'K':
    case 'k':
if(OnMyOwn) return 0;
if(DesiredVel > -MAX_VEL) sfSetVelocity(DesiredVel -= VEL_INCR);
return 1;
    case 'L':
    case 'l':
if(OnMyOwn) return 0;
if(DesiredRotVel > -MAX_RVEL) sfSetRVelocity(DesiredRotVel -=RVEL_INCR);
return 1;
    case 'H':
    case 'h':
if(OnMyOwn) return 0;
sfSetRVelocity(DesiredRotVel = 0);
return 1;
    default:
return 0;
  }
}

int main(int argc, char **argv)
{
  if(argc != 6) {
    cerr << "Usage: " << *argv << " LLN SNN HNN DAO DT" << endl;
    return -1;
  }

  SetupEnviron();
  SetupInterface(argv);
  SetupBasicConcepts();

  //start up Saphira asynchronously
```

```
  sfStartup(1);
  //sfKeyProcFn((int (*fn)()) myKeyFn);
  sfKeyProcFn(myKeyFn);
  sfSetDisplayState(sfDISPLAY, 0);  // we don't need no steenkin' display
  sfMessage("Awaiting connection...");
  while(!sfIsConnected) sfPause(100);
  sfMessage("Awaiting connection stabilization");
  sfPause(2000);

  for(;;) {
    if(OnMyOwn) Act();
    else Train();
    cout << '.' << flush;
    if(sfIsExited || !sfIsConnected) break;
  }
}
```