Representation Operators and Computation

Brendan Kitts

Department of Computer Science,
Center for Complex System,
Brandeis University, Waltham, MA. 02254. USA.
Email: brendy@cs.brandeis.edu

Abstract

This paper derives results about the impact of representation and search operators on the Computational Complexity of a problem. A model of computation is introduced based on a directed graph structure, and representation and search are defined to be respectively the vertices and edges of this graph. Changing either the representation or the search algorithm leads to different possible graphs, and hence have different complexity classes. The final section explores the role of representation in reducing time complexity in Artificial Intelligence.

Keywords: representation, redescription, recoding

1.0 Introduction

In solving real-world problems, a significant part of the task is finding a good encoding scheme. Another is finding an algorithm which manipulates the data efficiently. While the second enterprise (algorithm design) has been analysed in Computer Science, the first, selection of a representation, has been ignored or grouped with algorithm design.

This paper tackles the issue of representation, what it is and how it affects computation. A simple model of computation is defined which allows some straightforward properties about the power of search and representation operators to be proved. The second part of this paper discusses the role of representation in another computational area where representation may be the key, Artificial Intelligence.

2.0 A Graph Model of Computation

Many different formalisms have been used to define computation, among these Herbrand-Godel Computable, Markov computable, Abacus computable, Effective computability, and Turing Machine. This paper will introduce a new but equivalent model based around graph theory.

Definition 1: Connected Chain

A Connected Chain is a graph $G = (S,T)$ where the vertices S form the set of states $\{s_1,s_2,s_3,...\}$, and the arcs $T = \{(S \times S)\}$ form a set of unlabelled transitions from one state to another. G also has the following properties:

1. $S \neq \{\}$. There must be at least one state in the machine.

2. $s_{initial} \in S$. There must be a start state.

3. If $s_{accept} \in S$ then s_{accept} has outdegree 0. s_{accept} is called a final or halting state.

4. $\forall s \in S$ the maximum outdegree is 1. This forces the computation to be deterministic. An example of a Chain is given in figure 1.

§

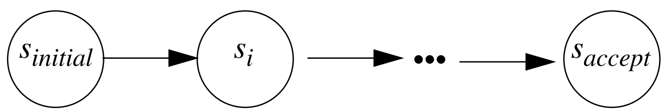


Figure 1: A Connected Chain. $S = \{s_{initial}, s_i, \ldots, s_{accept}\}$, $T = \{(s_{initial}, s_i), \ldots, (\ldots, s_{accept})\}$

<u>Theorem 1:</u> Any Turing Machine operating over an input can be simulated by a Connected Chain.

Proof

A Turing Machine is defined formally as a system $M = (Q,S,d,q_0,q_f)$, where Q = set of states, S = tape alphabet, d = transition function $Q \times S \rightarrow Q \times S \times \{L,R\}$, q_0 is the initial state, and q_f is the final state.

Any Turing Machine can be simulated by a Universal Turing Machine with 5 finite control states and a tape alphabet of $\{0,1\}$ (Hopcroft and Ullman, 1979). Let M be such a machine. An instantaneous state description of *M* can be encoded as a string *Tcode* consisting of the following parts:

read-head = Read-head position on the tape, coded as a unary integer.
fc = Finite control state, coded as a unary integer.
tape = A tape configuration variable which encodes all of the tape's contents. The tape variable codes the Turing tape in the following way. Since the alphabet consists of only two letters, we can encode the tape as the unary representation of a binary integer.

The complete *Tcode* is of the form $2^{rh}3^{fc}5^{ta}$. This code gives a unique description of the instantaneous state of *M*. We will now show that *M* can be re-described as a series of instantaneous states connected by unlabelled deterministic transitions.

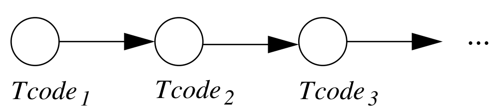


Figure 2: A Connected Chain is simply a series of Turing Machine instantaneous desecriptions connected by unlabelled arrows. $S = \{Tcode_1, Tcode_2, Tcode_3, ...\}$, $T = \{(Tcode_1,Tcode_2),(Tcode_2,Tcode_3), ...\}$

For any Turing Machine, the following semi-effective procedure can be used to convert the machine into a Connected Chain:

<u>Turing Machine program:</u> Convert <TM,input> to Connected Chain

Start at initial state, q_0, and the initial tape contents (the input).

Recode the current *(fc,tape,read-head)* as a state $s_{source} \in Tcode$.

$s_{initial} = s_{source}$

$T=\{\}$

$S=\{s_{initial}\}$

Simulate the Turing Machine step by step. For each step

If there is no *d* defined for this state, then $s_{accept} = s_{source}$ and Halt

Perform *d:(fc,read-head)* $\rightarrow$ *(fc,read-head,{L,R})*

Encode the new Turing Machine state as $s_{new} \in Tcode$

$T = T \ \mathbf{U} \ (s_{source}, s_{new})$

Search through list of existing states in the Chain. If there exists an $s_{old} = s_{new}$ which has already been encountered then Halt.

$S = S \ \mathbf{U} \ \{s_{new}\}$

$s_{source} = s_{new}$

continue until the TM halts.

§

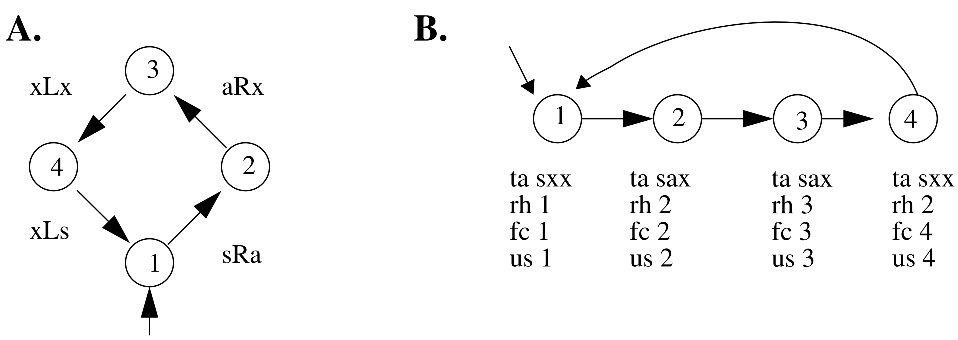


Figure 3A: non-halting Turing Machine which loops infinitely. **B:** The same machine represented as a Chain.

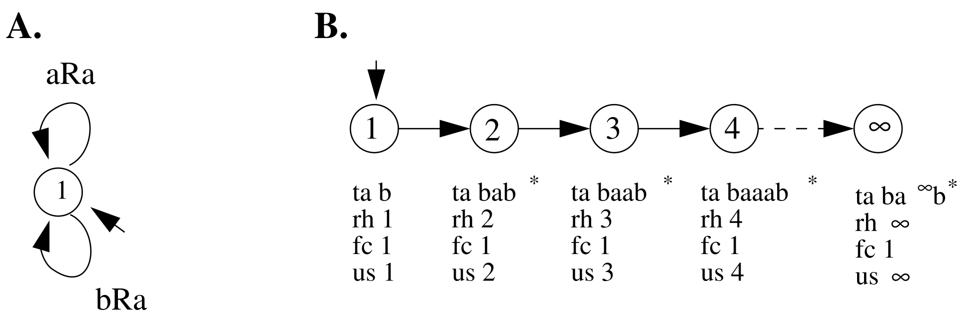


Figure 4A: Turing Machine which uses an infinite amount of tape. **B:** a sketch of the Chain. The Turing Machine works as follows: Given a tape initialised to all blanks, the Turing machine at left reads a blank, moves right and writes an a each step forever thereafter. The result is a tape filling up with as, and an infinite number of vertices in the Chain.

Definition 2: Unconnected Chain

An Unconnected Chain *G'* is a set of Connected Chains. The Connected Chains comprising *G'* may be unconnected, or may "run into each other" as shown in figure 5. An Unconnected Chain must satisfy the properties of a Connected Chain. §

Theorem 2: Turing Machine → Unconnected Chain

Any Turing Machine with unspecified input can be simulated by an Unconnected Chain

Proof

For a non-deterministically chosen input, run the Convert <TM,input> to Connected Chain procedure. §

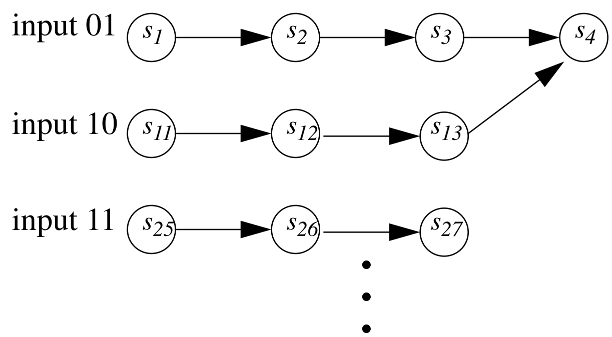


Figure 5: An Unconnected Chain simulating a Turing Machine. An Unconnected Chain is a union of Connected Chains. $S = \{s_1,s_2,s_3,s_4\}$ U $\{s_{11},s_{12},s_{13},s_4\}$ U $\{s_{25},s_{26},s_{27},s_4\}$ U ..., $T = \{(s_1,s_2),(s_2,s_3),(s_3,s_4)\}$ U $\{(s_{11},s_{12}),(s_{12},s_{13}),(s_{13},s_4)\}$ U $\{(s_{25},s_{26}),(s_{26},s_{27})\}$ U ...

Theorem 3: Unconnected Chain $\rightarrow$ Turing Machine

Proof

A semi-effective procedure for building a Turing Machine which simulates a *Connected* Chain is given below:

Turing Machine program: Convert Connected Chain to Turing Machine

Let $ta=\{\},rh=1,fc=1$.

Let $q_0=1,\ q_f=\{\}$

Let $s_{source} = s_{initial}$

$encountered=\{\}$

Repeat: step through the Chain

$encountered = encountered\ \ \boldsymbol{U}\ (s_{source}, fc, ta, rh)$

If $\exists s_j \bullet (s_{source}, s_j) \in T$ and $(s_j,x,y,z) \notin encountered$ then $d = d\ \boldsymbol{U}\ \{((fc,B), (fc,1,\{R\}))\}$

If $\exists s_j \bullet (s_{source}, s_j) \in T$ and $(s_j,x,y,z) \in encountered$ then run subroutine which resets the T uring Machine to state x,y,z and Halt.

if $\neg\exists s_j \bullet (s_{source}, s_j) \in T \wedge s_{source} = s_{accept}$ then $q_f = fc$, Halt

Let $s_{source} = s_j$

continue

This algorithm simply writes a 1 on each square of the tape and moves right, until the end of the chain is reached. If a loop is encountered, it returns the Turing Machine to the state it was in at the beginning of the loop. It does this by saving a look-up table of state *Tcode* labels used in the Chain and the *(fc,ta,rh)* state of the Turing Machine. This lookup table can be implemented on a separate tape. The subroutine for reseting Turing Machine state works by reading this separate tape, wiping the current Turing Machine contents, and adding 1s to the tape according to the number specificed in the stored tape location.

To simulate an Unconnected Chain, nondeterministically run the Convert Chain to Turing Machine algorithm on $Chains \in G'$. §

Like the Fourier transform, the Chain representation enhances some aspects of the process/computation, notably the time aspect, whilst making others more difficult, notably the space aspect. Space complexity is defined as the maximum size of tape used by a Turing Machine. In a Chain the tape contents are encoded as part of each state (node), and there is no way to tease apart the tape aspect of each state from the other aspects, therefore space complexity becomes a quantity which can't be measured. Time Complexity on the other hand becomes very clear, simply defined as follows:

Lemma 1: The Time complexity of G' is equal to the minimum number of nodes from $s_{initial}$ to s_{accept} in G'.

Proof

Time complexity is defined as the minimum time taken for a Turing Machine to halt (Hopcroft and Ullman, 1979). Since a transition from vertex s_1 to s_2 is equivalent to a transition in a Turing Machine, then the number of nodes from $s_{initial}$ to s_{accept} give the number of steps taken in the Turing Machine from its starting to halting state. The minimum number of nodes between $s_{initial}$ and s_{accept} in G must therefore define the time complexity of the computation. §

The Chain formalism also has some drawbacks. Firstly, in the Turing Machine and other classical models, program and input are separated. Proofs such as Blum's Speed-up theorem concern a particular machine with an unspecified input tape. In this model, program and input are modelled together as the one system and we loose the distinction. A second disadvantage is that infinite processes require an infinite sized graph to capture, where-as Turing Machines, finite automata, and others have notational mechanisms to neatly represent unbounded languages.

The usefulness of the Chain is that it is extremely simple and ignores stored-program machinery (the tape and program), and emphasises the state-update aspect of computation. The Chain is so simple that it can easily be mapped to many other classical Formal Systems.[1] Another advantage of the Chain is that it allows a simple and intuitive definition of search and representation, and shows that both together make up what will be called a *Domain Computation*.

3.0 Domain Computation

The preceeding definition of a Chain is what might be called a computing machine and/or program which doesn't actually solve a problem. In solving problems however, we need to develop a convention system to map from states in the machine to objects in the external world, so that our inputs and outputs solve a problem. This mapping is called a *representation*. The existence of such a representation has significant computational consequences which have not been addressed in classical Computing Theory. First we define search, then representation.

Definition 3: Search algorithm

A Search algorithm will be defined as the transition operator set $T = (S \times S)$ of a given Computation. §

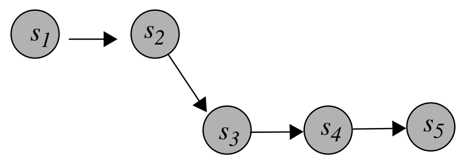


Figure 6: *The Search algorithm of a computation* is defined as the set of transition operators over G. The search algorithm for the above graph is equal to $\{(s_1,s_2),(s_2,s_3),(s_3,s_4),(s_4,s_5)\}$.

Definition 4: Representation

Let O be the set of states in the world at large which an observer is interested in. A representation is defined as a mapping $R:O \rightarrow S$. This is a mapping from objects in the world which the user is interested in, to states of a machine. A representation can be 1:1, 1:M, M:1, or M:M[3]. An additional constraint is that $O \neq \{\}$, dom($s_{initial}$)$\neq \{\}$, and dom$(s_{accept})\neq \{\}$ if the computation halts. §

Representation is identical to the *domain theory* or *model* for a formal system. It provides a semantic mapping which instantiates what the states of the machine mean. Once a representation is defined, the computation from the observer's standpoint traverses the representation space.

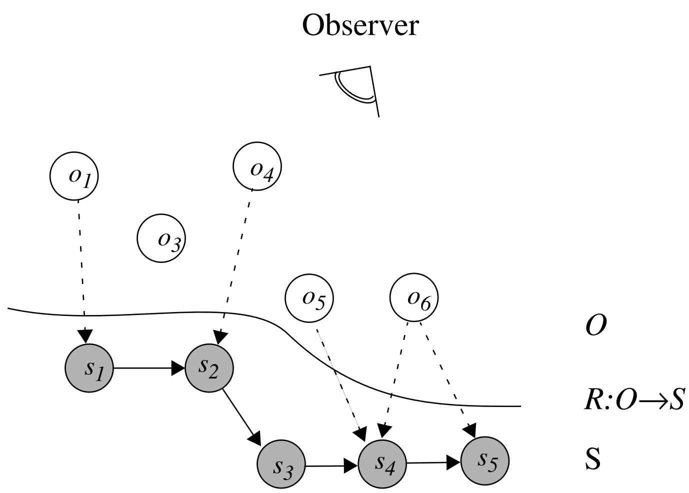


Figure 7: A Representation relation maps objects in the world to underlying computer states. Here s4 and s5 states in the machine are considered the same to the observer.

<u>Definition 5:</u> Domain Computation

A *Domain Computation* is defined as any computation where a representation is defined. §

To summarise, as a result of these definitions we have computation = directed graph G, Domain Computation = directed graph defined over a observer's encoding, representation = a mapping from O to S, or from the observer's perspective, the objects in his/her world which the graph traverses, and search algorithm = edges of G, or from the observer's perspective, the order in which the objects in his/her world are traversed.

4.0 Changing Computational Complexity

This section deals with the question of how a computation can be changed for the better. Assume that we have a program which computes some function, such as the profit and loss for a company. Is there a way to improve this computation, where improvement means making the computation occur in less time, making the computation more accurate, or use less space? In this section it will be shown that search and representation seem to be complimentary, in that from an initial computation graph *G*, both together allow all possible graphs to be reached. Therefore both representation and search operators can have unique influences on problem complexity. The nature of those influences will be discussed in this section.

Definition 6: Search change and Representational change on graph *G*

Let a search change be the selection of any new transition operator, *T'*, and representational change, or a representational recoding), be the selection of any new representation mapping, *R'*. Search and representation changes are generated by search operators $Sop:G \rightarrow S'$, and representation operators, $Rop:G \rightarrow R'$. §

Search and representation operators are actually themselves computations. A search operator can be described as a Turing Machine which takes an encoding of a Turing Machine *G* on its tape, and generates a new encoding of *G*, where the transition set of *G* (defined in graph theory above) has changed

The Representation operator can't affect the external object mapping imposed by the observer, since this is just a semantic mapping created outside of the computation. However, it can operate on the underlying set of states *S*, by mapping sets of states, $\{s_1,s_2,..,s_n\}$ into a new state, s_j, or a single state s_j into a set of states $\{s_1,s_2,...,s_n\}$. By changing the underlying state set *S* which the transition set operates over, the representational recoding changes the relation $O \rightarrow S$ and hence the representation. This also results in a nice dualism, search operators operate over T and representation operators operate over S, and we originally defined a computation as $G=(S,T)$. Thus, analogous to search operators, a representational operator takes a description of the Turing Machine, and writes a new description of a Turing Machine, where the state set S in the new machine is different.

<u>Lemma 2.1:</u> Possible Graph effects resulting from representation change

Consider any Chain G which comprises some computational device. The four possible effects of a representational recoding R' applied to G are defined below. These will be referred to as the set of Graph remappings.

- If R' is 1:1 with R, then the graphs possible are equal to all topologically isomorphic graphs of G, but with each node being a unique relabelling of G.

- If R' is 1:M with R, then graphs identical to G can be generated, but with new dummy nodes are interspersed between existing transitions in graph G.

- If R' is M:1 with R, then graphs identical to G can be generated, but with nodes deleted, and the transitions to and from each deleted node, n, (x,n) and (n,y) replaced by a single transition (x,y).

- If R' is M:M with R, then a new graph can be generated which is the union of the transforms above, comprising graphs of G with nodes deleted, nodes inserted, or node order re-arranged. §

<u>Lemma 2.2:</u> A Representational mapping cannot form a cyclic graph from an acyclic G, nor can it form an acyclic graph from a cyclic one.

Proof

2.2.1: Linear chain to cycle not possible

None of the possible representation mappings change a linear chain into a cycle, and therefore, recoding cannot change a linear chain into a graph with a loop.

2.2.2: Cycle to linear chain is not defined for all cases

If G is a chain with a single cycle, then any 1:1 recoding must also result in a chain with a single cycle, since the recoded graph is isomorphic to G. Insertion (1:M) doesn't affect the cycle. Deletion (M:1) can remove the cycle in some cases - by the determinism property, there can only be one cycle in G, and this cycle must flow from node N to some node earlier in the chain, $i \leq N$. If this cycle flows from node N to any node earlier in the chain $i>1$, then the cycle can be removed by deleting all nodes from i to N inclusive. However, if $i=1$, then there is no way to remove the cycle. Removing 1 to N would cause $S=\{\}$ which is illegal (Definition 1). Since the representational operations cannot break the cycle for all cases, a representational recoding cannot arbitrarily build a linear chain from a chain with a cycle. §

Lemma 2.3: Search change cannot increase the number of states traversed by a computation beyond the states defined in S.

Proof

This is true because the number of edges are always less than or equal to the number of vertices in the graph. §

Definition 7: Recursive and Recursively Enumerable Graphs

A recursive graph G_r is a graph of a computation which is recursive. A recursively enumerable graph ("r.e. graph") $G_{r.e.}$ is a graph of a computation which is recursively enumerable. §

Theorem 5: A recoding *R'* cannot convert an r.e. graph into a recursive graph. However, a recoding *R'* can convert a recursive graph into an r.e. graph.

Proof

From Lemma 2.2 a recoding cannot change whether a graph halts or does not halt. Therefore, a recoding cannot change a graph which is r.e. to a graph which definitely halts.

Because a recoding algorithm is itself r.e., it is also undecidable as to whether the graph it generates contains a halting state. Therefore, a recoding algorithm can convert a recursive graph into an r.e. graph. §

Theorem 6: A search change *T'* can convert an r.e. graph into a recursive graph. However, a search change *T* cannot convert a recursive graph into an r.e. graph.

Proof

Given an r.e. graph, a search change T' can simply replace all of the edges, T, with a single edge $T'=(S_{initial},S_{initial})$ (or any other desired finite set of edges).Therefore, an r.e. graph can be changed to a lower complexity graph.

Given a recursive graph, it is not possible for a search change *T'* to create an r.e. graph. The reason is that the number of edges are less than or equal to the number of vertices in the graph (Lemma 2.3). Therefore, if the number of vertices are finite (which must be so in the case of a recursive graph), then the number of edges which must be less than or equal to this number, and so it is not possible for a finite graph to be turned into an r.e. graph by changing *T*. §

<u>Theorem 7:</u> Completeness of search and representation operators

It is possible to generate all possible graphs, and hence all possible Turing machine computations (theorem 1) by using search and representation changes. (theorems 6 and 7). §

<u>Discussion:</u>

Theorems 5 and 6 show that representation and search are complimentary to each other. Representational recoding can change a recursive graph into an r.e. graph, while search operators can change an r.e. graph into a recursive graph. Both operations (with different selections of $o_{initial}$ and o_{accept}) together allow for the complete range of possible classes of computations to be reached from an initial computation.

A final simple theorem will show that representation and search can both minimize the time complexity of recursive problems.

<u>Theorem 8:</u> If a computation is recursive, there exists a recoding which minimizes time complexity of the computation.

Proof

Because the graph is finite, a recursive procedure can map every initial state which ultimately ends in an accepting state, $s_{initial}$ to the state s_{accept}, and every non-accepting s_x to s_{reject}. Running the same computation over *S'* now solves the problem immediately. §

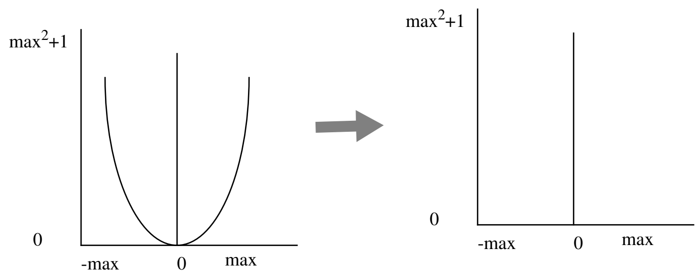


Figure 8: Grefenstette's "needle in a haystack" optimisation problem. This problem can be optimally recoded by defining a recoding algorithm which maps all x values to x=0.

Theorem 9: (Jones, 1995) If a computation is recursive, a search algorithm exists which minimizes the time complexity of the computation.

Proof

Simply reconstruct graph *G*, with a transition from each initial state to the accepting state, and from each non-accepting state to a non-accepting node. The problem is solved in one step. §

This completes the duality on time complexity for recursive computations. A problem can be solved in minimum time by either finding an optimal search algorithm, or by searching the space of representations.

5.0 Recoding and Artificial Intelligence

Human beings face immense complexity in everyday life, and yet exhibit impressive real-time performance. Human cognitive responses generally span a range of

50 to 200 miliseconds (Churchland and Sejnowski, 1992). Axon conduction velocity is between 0.6m/s and 120m/s (Chudler, 1997). Let's assume signals travel 2 meters to their targets, and then 2 meters back to their effectors. That gives 33ms in travel time. Assume that there are 5 interneurons on afferent and efferent pathway respectively. Membrane integration time is about 7ms (Sah and Bekkers, 1996). Therefore membrane integration consumes 70ms. This leaves on the order of 100ms for computation in the central nervous system. Any computation in the CNS will incur a large 7ms overhead for each layer of neurons mapped to, meaning that there could be as little as 100ms/7ms = 14 "steps" allowed in the brain proper.

The kinds of computations which could be computed by a 14 step architecture would seem to be very limited. With little time to perform search, and a vast array of functions which need to be implemented accurately (life depends on it), the only possibility open to the brain seems to be the massive use of space to defeat time complexity.

Instead of computing a detailed solution to every problem, the brain could instead operate by storing solutions ready-computed and then activating them when needed. Architectures which support a one-step style process will have a look-up table structure, candidate systems currently including content-addressable memory, feedforward neural networks, hash tables, linear transformations and adaptive filters. We will refer to these as maps following the formalism in category theory. Instead of handling different functions by some clever computation or transformations, we might theorize that they are just handled by different maps[3]. The picture that emerges is one of a series of maps, each layer of which provides a different or more refined recoding of a problem, where the information extracted at any level can be used.

There is neurological evidence which supports the idea of multiple representations, or maps, in the brain. In addition to detectors for primitive features such as edges, the brain also has detectors for amalgams of these features, complex objects such as hands and faces (Hubel, Weisel and Stryker, 1978). Visual, auditory, and other topographic maps are also found at multiple places all over the cortical surface (Kandel, Schwartz and Jessel, 1992). The central issue in *The Imagery Debate* (Tye, 1991) was why is there any need to "re-detect" stimuli at new areas in the brain after they have been registered at the periphery. More bluntly, why would the brain need to have pictures inside the head? For a homunculus? The debate was fueled by a predominance of experimental evidence which actually supported the existence of all these internal "pictures"! (Kosslyn, 1994). This paper suggests that these phenomena might exist because there is a time and complexity advantage in maintaining multiple, redundant, partially processed recodings. Stimuli detected by the retina is partially transformed at the visual cortex, and then partially transformed again at higher level areas VII, VIII, IT. At each level, the representation is useful. Early pre-processed visual representations are wired to efferent fibres so that even primitive recognition can trigger action quickly. These representations lower the time complexity of the overall behavioural problem.

Multiple stimulus-like remappings are also probably seen because the brain had to evolve, and could only change incrementally. Radical rewirings couldn't be acheived instantly, so the brain shows the development of gradual transformations, each minor change of which had to be useful. Eventually, these abstract representations might have provided a means for yet more sophisticated computations, because the underlying representations generated a more powerful language (representation) for building more complex computations. An example of this phenomenon is provided by the decision tree system AQ17-HCI (Wnek and Michalski, 1994), which recodes subtrees into new primitives and re-induces tree with the more abstract language. This allows the tree to model relationships which were not possible with the original language[4].

Artificial Intelligence systems have either explicitly or implicitly (normally implicitly) made strong assumptions about the kind of representation required. The results have been equivocal. Expert Systems, a kind of symbol system, exhibited impressive performance in narrow knowledge areas but were largely unable to venture outside these knowledge areas. Neural networks on the other hand have been powerful in low-level tasks. However, learning multiplication tables or instances of Ohms law do not come naturally for these systems - they learn the literal number relations, not the general law (Clark, 1993; Pinker and Mechler, 1988; Goldfarb, 1994).

Although both symbolic and sub-symbolic schools have developed elegant and highly successful techniques, neural nets and expert systems individually cannot handle the broad set of problems which living organisms must deal with. The reason is that neural network representation exists at too primitive a level to successfully represent high-level relations, and converse problems plague expert systems such as brittlness, domain narrowness, and sensitivity to input noise.

The search to find The “correct” Language of Thought (Fodor, 1974) has been fraught with problems. Cognitive theories, have ranged from the first-order logic systems of McCarthy (1968), the enhanced first-order system under development by Lenat (1995), to the recent, although more biologically inspired claims by the neural network community (Rumelhart et. al, 1986). The view maintained by this paper is that no single representation system will be ideal for all cognitive tasks, and rather the key is that many specialised representational recodings, or coding variants are needed. This strategy decreases the time complexity dimension of the task (by increasing space complexity), and leads to competance on a broad range of problems rather than just the tasks the one representation happens to facilitate.

6.0 Related work

Early work in Artificial Intelligence showed sensitivity to the problem of representation (eg. see Amarel, 1968 pp. 164-167). However, it has not been until recently that machine learning practicioners have begun to use representation operators in a practical sense.

In Genetic Algorithms, Holland's inversion operator (Goldberg and Bridges, 1990) takes bits on a bit-string and simply re-orders them in the hope of reducing the tendency of a GA to become trapped by artifacts of the bit encoding. Kingdon and Dekker (1995) (also Toth, et. al., 1995) suggest a representation operator for Genetic Algorithms which automatically switched number base from binary to base 3, 4, or higher, when a problem appeared to be improperly converged. Koza has been highly successful with the use of Automatically Defined Functions or program modules, which embody a large sub-program and can be used in further genetic search. Other innovative work in representation has come from the Decision Tree Induction field. Systems such as TGCI (Donoho and Rendell, 1995) and AQ17-HCI (Wnek and Michalski, 1994) look for complex patterns in a decision tree which appear to be "repeated", and then recode this as a new attribute or feature.

Theoretical aspects of representation have been touched on by Kolen and Pollack

(1995) and Crutchfield (1993). Kolen and Pollack showed that the complexity class of processes was relative to the representation chosen to view it. These authors provide an example of a chaotic process which could be converted into a deterministic process by simply sampling at a coarse level of abstraction. The idea of complexity being relative to point-of-view also leads to the conjecture that the apparent complexity of Natural Language may be an artifact of the representation used - words and sentences.

Clark and Thornton (1997) have presented a *Brain and Behavioural Sciences* target article in which they suggest that trading one problem space for another can be used to solve problems. They also present some function learning problems in which a change in representation makes the problem tractable.

Finally, a Psychological aspects of representational recoding have been advanced by Annette Karmiloff Smith (1992). Karmiloff-Smith suggested that children go through various development shifts which are caused by the redescription of their knowledge in new ways, for instance, from procedural to declarative knowledge. She presents interesting evidence of sudden cognitive shifts where children go from being locked in to certain behaviours, to having mastery over them. Boden (1994) has also suggested that the ability to create a new conceptual space is the foundation of human creativity.

10.0 Conclusion

This paper has addressed the question of how representation affects computation. It has been shown that

1. Computation can be redefined as a special directed graph called a Chain.
2. Search and representation operators - programs which modify each of these aspects of computation - comprise a complete set of operators on computation. Some computation changes can be achieved with representational operators that *cannot* be with search operators and vica versa.
3. Multiple representations can be used to lower the time complexity of computation. Representational recodings might also be an important factor in the development of broad competance rather than narrow expertise.

Representation has been something like a “black art” in Artificial Intelligence. The state of the art today is for a designer to use human expertise and insights about a problem to find a good representation from which a learning algorithm or optimization method can then solve the problem. The use of representation operators has not been adopted widely as yet. The analysis of this paper shows that representation has important and non-trivial affects on the Computational Complexity of a problem, some of which in theory *cannot* be replicated by changing heuristic state-update algorithm. The implementation of representation warping algorithms is a key strategy in the quest for a general purpose intelligent system.

Acknowledgements

Endnotes

1 Lemma 2.2: Formal System $\rightarrow$ Chain

This theorem shows that a Formal System can be implemented by a Chain. The corollary is that the computational properties of Chains can be used to analyse a Formal System.

Proof

A formal system is a system which has the following components:

(i) reachable state space or wffs, which is a set of states which the system could be in;
(ii) axioms or initial states;
(iii) state transitions, update equations, or rules of inference, which is a mapping between one state and another
(Mendelson, 1964).

Let the vertices of G, $s \in S$ represent the wffs, let arcs $t \in T$ represent inference rules from one wff to another, let $\{s_{initial}\} \subseteq S$ represent initial state(s). Formal Systems respect the determinism principle. Therefore a Chain can be defined for any Formal System.

Corollary 2.1: For any Formal System we can calculate the Computational Complexity of that system.

Since TM$\leftrightarrow$Chain, Given a Formal System, simply calculate the Computational Complexity of the Turing Machine resulting from the Chain which is implementing the Formal System.

2 Examples of representations which are 1:M, M:1, and M:M:

One to many: Many programs may represent the one language. In nature, RNA primary and secondary structure is highly redundant.

Many to one: Genetic Algorithm schemas represent templates of possible solutions, so too Branch and Bound partial solutions. CMACs and Kohonen nets store and manipulate prototypes which represent a much larger set of objects.

Many to many: DNA is both distributed and redundant. A single gene can encode several phenotypic traits (pleitropy) and a single phenotypic trait is affected by more than one gene (polytropy). (Jones, 1995).

3 Time is at a preimum only for execution (and is presumably why myelinated neurons were evolved in the first place - the need for rapid signal transfer). However for learning the brain could use much slower processes, particularly chemical processes and neural RNA changes for effecting morphological changes to membrane structures and synaptogenesis.

4 Karmiloff Smith (1992) has reported that if asked to “draw a two-headed man”, 5 year olds will draw the first man, start on the second head, but then mistakenly complete a full man. Reportedly, children will try repeatedly to draw the two headed man, getting frustrated each time. It is as if the system is trapped at some local minima, and cannot itself break out to find a correct mode of behaviour. Karmilloff-Smith suggests that the dramatic cognitive shift which occurs around this age, where children can learn to do this task is associated with a theory called *Representational Redescription*. She theorises that children’s changing capabilities are the result of an “endogenous drive to redescribe knowledge in more abstract or general ways”, (ibid., 1992), and proposes that children’s representations are driven to change implicit (level-I) representations to varying degrees of explicit representation (E1, E2/3).

References

Boden, M. (1994), ‘Précis of the Creative Mind’, *Behavioural and Brain Sciences*, Vol. 17, pp. 519-570.

Chudler, E. (1997), Brain Facts and Figures, http://weber.u.washington.edu/~chudler/facts.html#neuron

Churchland, P. and Sejnowski, T. (1992), *The Computational Brain*, MIT Press, MA.

Clark, A. and Thornton, C. (forthcomming), 'Trading Spaces: Computation, Representation and the Limits of Uninformed Learning', *Brain and Behavioural Sciences*. Also available as Technical Report, Washington University, St. Louis, MO.

Crutchfield, J. (1994), 'The Calculi of Emergence: Computation, Dynamics, and Induction', *PhysicaD*, Vol. 75, pp. 11-54.

Donoho, S. and Rendell, L. (1995), 'Rerepresenting and Restructuring Domain Theories', *Journal of Artificial Intelligence Research*, Vol. 2, pp. 411-446.

Goldberg, D. and Bridges, C. (1990), 'An Analysis of a Reordering Operator on a GA-Hard Problem', *Biological Cybernetics*, Vol. 62, pp. 397-405.

Goldfarb, L., et. al. (1994), 'Can a Vector space-based learning model Discover Inductive Class Generalisations in a Symbolic Environment?' in *Proceedings of the 10th Biennial Conference of the Computer Society for Computational Study of Intelligence*, Morgan Kaufmann, CA.

Hubel, D., Wiesel, T., and Stryker, P. (1978), "Anatomical Demonstration of Orientation Columns in Macaque Monkey", *Journal of Computational Neurology*, Vol. 177, pp. 361-380.

Jones, T. (1995), 'Evolutionary Algorithms, Fitness Landscapes and Search', Doctoral Dissertation, University of New Mexico.

Kandel, E., Schwartz, J., and Jessell, T. (1992), *Principles of Neural Science*, Appleton and Lange. CI.

Karmiloff-Smith, A. (1992), *Beyond Modularity*, MIT Press.

Kingdon, J. and Dekker, L. (1996), ‘The Shape of Space’, Technical Report, Department of Computer Science, University College London.

Kosslyn, S. (1994), *Image and Brain: The Resolution of the Imagery Debate*, MIT Press.

Lenat, D. (1995), ‘CYC: A Large-Scale Investment in Knowledge Infrastructure’ *Communications of the ACM*, Vol. 38, No. 11, pp. 32-38

McCarthy, J. (1968), ‘Programs with Common Sense’, in Minsky, M. (ed), *Semantic Information Processing*, MIT Press, MA.

Mendelson, E. (1964), *Introduction to Mathematical Logic*, Van Nostrand Company, New York.

Pinker, S. and Mechler, J. (eds), (1988), *Connections and Symbols*, MIT Press, MA.

Radcliffe, N. (1994), ‘The Algebra of Genetic Algorithms’, *Annals of Maths and Artificial Intelligence*, Vol. 10, pp. 339-384.

Rumelhart, D., McClelland, J., et. al. (1986), *Parallel Distributed Processing*, Vol. 2, Mit Press, MA.

Sah, P. and Bekkers, J. (1996), “Apical Dendritic Location of Slow Afterhyperpolarization Current in Hippocampal Pyramidal Neurons: Implications for the Integration of Long-Term Potentiation”, *Journal of Neuroscience*, Vol. 16, No. 15, pp. 4537-4542.

Toth, G., Kovacs, S. and Lorincz, A. (1995), ‘Genetic Algorithm with Alphabet Optimisation’, *Biological Cybernetics*, Vol. 73, pp. 61-68.

Tye, M. (1991), *The Imagery Debate*, MIT Press.

Wnek, J. and Michalski, R. (1994), ‘Hypothesis-driven Constructive Induction in AQ17-HCI: A Method and Experiments’, *Machine Learning*, Vol. 14, pp. 139-168, Kluwer Academic Publishers.

APPENDIX

Examples of Recoding in Machine Learning

Problem Solving Games

A classic problem in AI is the 8-queens; a search problem where the problem is to place 8 queens on a chess board such that none attack each other (from Maly and Hanson, 1979). The intuitive solution is to represent the board as an eight-by-eight array and try queens in each possible board position, has a search space of $64*64-1*64-2*...*64-q$ where $q=8 = 9.9\times10^{15}$ possible states. However, it was found that a representation which incorporates the idea that queens should be placed on separate horizontals reduces the search space to only $8*8*...*8$ q times = $8^q = 1.7\times10^7$ states. The difference between the two approaches is that in the second domain knowledge allows the creation of a better problem representation, reducing problem complexity and allowing better performance.

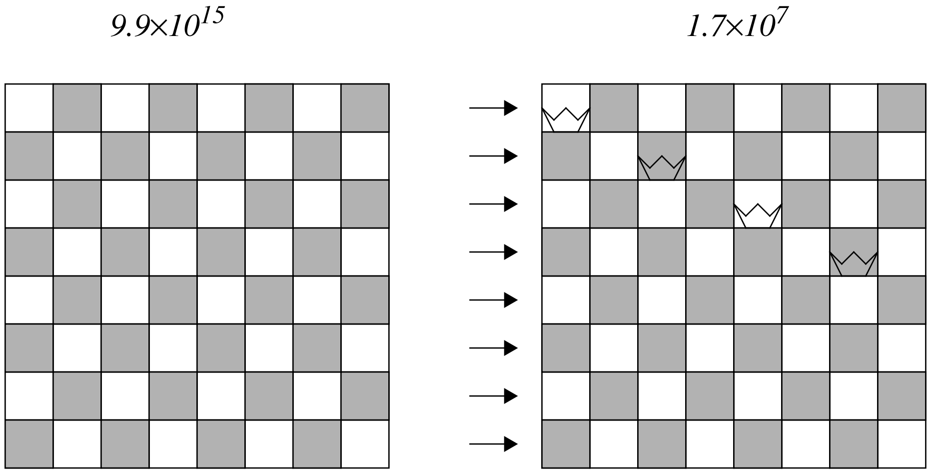


Figure 9: A different representation which has queens only appearing on horizontals can convert a brute-force n-queens problem into a problem with a much reduced search space, and so faster computation.

Optimisation

Optimization refers to the process of finding the minima of a function. Algorithms often use a rule known as a "hill-climbing", which works by testing for the payoff along each variable dimension, and then proceeding in the direction of best payoff. Basic hill-climbers traditionally have problems (among other things) with "ridge" formations; long narrow regions where the search becomes "trapped" by being unable to follow the run of the ridge, since they may only move parallel to axes (figure 10). This kind of problem can be ammeliorated to a large extent by stretching the axes so that the problem becomes more symmetrical. Stretching the axis is "recoding" because new states in the device are defined to represent the same object / landscape topography which was previously used in the computation.

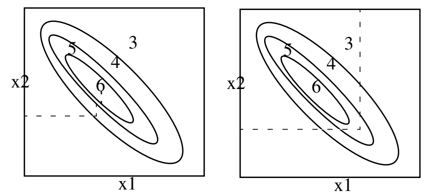


Figure 10: A ridge problem. The hill climbing search can only move in a direction parallel to each of the axes, so in the second example it runs into a ridge and cannot move any further. Isolines show fitness, and the dotted line shows the trajectory of the search.

Decision tree Induction

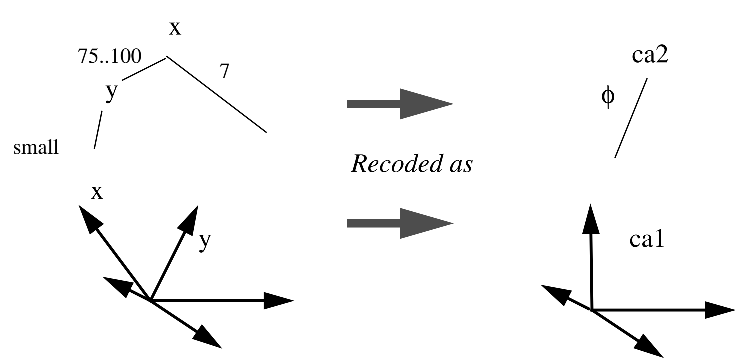
The AQ17-HCI algorithm (Wnek and Michalski, 1994) is a tree induction algorithm for classification problems. It uses a C4.5 like procedure for creating the tree, but in additional uses representational space transformations, referred to as a "hypothesis-based construction" (HCI) algorithm. Following induction of a decision tree "hypothesis" using AQ17, the HCI algorithm traverses the tree looking for particular patterns which it can combine to create new features in a representational language. Three patterns may be identified: value patterns (a group of attributes), condition patterns, and rule patterns (groups of features) (Figure 11).

HCI determines the strength of each pattern by calculating the number of positive and negative examples "covered" by the pattern. This is meant to give an indication of the degree to which the pattern/attribute/condition in the hypothesis matches the target concept. After identification of patterns and assignment of "strengths", the HCI algorithm manipulates the representation language in three ways:

1) **remove redundant symbols**; symbols which do not appear in any of the rules,
2) **remove insignificant symbols**; symbols which only appear in rules with low strength.
3) **construct new symbols** from high strength patterns.

The first two recodings have the effect of collapsing useless dimensions, reducing computational complexity and so efficiency of search. The latter creates a new "feature" or dimension in the search space. This new dimension is a collection of high strength features and better captures the phenomenon in question using a condensed alphabet.

A. *(x=75..100 and y=small) or x=7 ::= ca2 =* ϕ



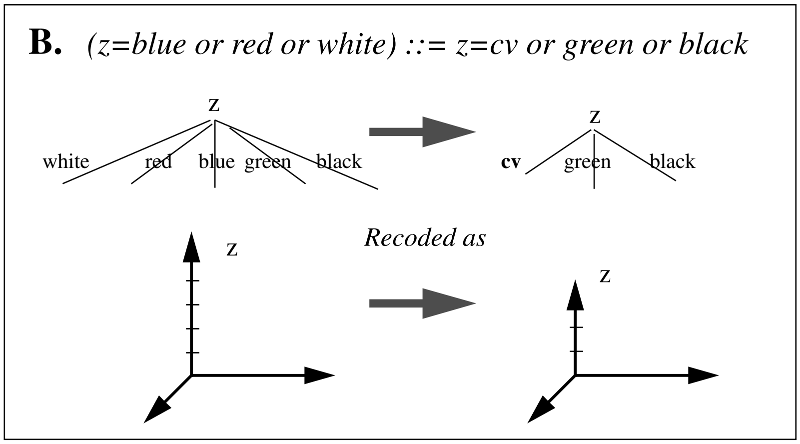


Figure 11: Examples of "language construction" mechanisms in HCI. The three parts of each figure correspond to (top to bottom) rule, decision tree, and state-space depictions of the process. **A.** A rule-pattern (top) is recoded as a new single attribute. This corresponds to collapsing a number of dimensions into one. **B.** A number of values of an attribute are recoded as just one value. These new features or modified attribute dimensions comprise a new representation language in which the problem is expressed. These are designed to improve solution quality.

The result is a search space which still retains the same regularities captured from previous attempts at classification, but which is far more easily searched (since expressions are traversed in "chunks"). (Donoho and Rendell, 1995).

References for Appendix

Craven, M. and Shavlik, J. (1995), 'Investigating the Value of a Good Input Representation', in Petsche, T. et. al. (eds), *Computational Learning Theory and Natural Learning Systems*, Vol. 3, MIT Press, MA.

Dartnall, T. (1995), ‘Redescription, Information and Access’, *Proceedings of the Third Australian Cognitive Science Conference*, Brisbane, Australia.

Dennis, J. and Schnabel, R. (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, NJ.

Elman, J.L. (1993), ‘Learning and development in neural networks: The importance of starting small’, *Cognition*, Vol. 48, pp. 71-99.

Haugeland, J. (1989), *Artificial Intelligence: The very idea*, MIT Press, MA.

Hertz, J., Krogh, A., and Palmer, R. (1991), *Introduction to the Theory of Neural Computation*, Addison-Wesley, Sydney.

Holland, J. (1992), *Adaptation in Natural and Artificial Systems*, MIT Press, MA.

Maly, K., and Hanson, K. (1979), *Fundamentals of the Computing Sciences*, Prentice-Hall, NJ.

Quinlan, J. (1993), *C4.5: Programs for Machine Learning*, Morgan Kauffman, CA.

Radcliffe, N. and Surry, P. (in press), ‘Fundamental Limitations on Search Algorithms’, to appear in *Lecture Notes in Computer Science*, Vol. 1000, Springer-Verlag.

Rendell, L. and Ragavan, H. (1993), 'Improving the design of induction methods by analyzing algorithm functionality and data-based concept complexity', *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Vol. 2, pp. 952-958, Morgan Kaufmann.

Revesz, G. (1983), *Introduction to Formal Languages*, Dover Publications.Turing, A. (1969), 'Intelligent Machinery', reprinted in Ince, D. (ed), *The Collected Works of A.M. Turing*, Vol. 3, pp. 107-131. North Holland.

Wellman, H. (1992), *The Child's Theory of Mind*. MIT Press, MA.

Wolpert, D. and Macready, W. (1995), 'No Free Lunch Theorems for Search', Technical Report SFI-TR-95-02-010, Santa Fe Institute.