In *AI Communications 7 (3/4)*, 203-219, 1994

COMPUTING AND INFORMATION COMPRESSION: A REPLY

J Gerard Wolff

June 1994

1.2

School of Electronic Engineering and Computer Systems, University of Wales, Dean Street, Bangor, Gwynedd, LL57 1UT, UK. Telephone: +44 248 382691. Fax: +44 248 361429. E-mail: gerry@sees.bangor.ac.uk.

Abstract

An earlier article [25] discusses the proposition that the storage and processing of information in computers and in brains may often be understood as information compression. A subsequent article [15] criticises the computing aspects of [25] and research on the more specific conjecture that all forms of computing and formal reasoning may usefully be understood as information compression.

The present article, which is intended to be intelligible without recourse to earlier articles, answers the main points in [15], tries to correct the many inaccuracies and misconceptions in that article, and discusses related issues.

Topics which are discussed include: the way theories are or should be developed; the role of evidence in motivating research; apparent shortcomings in the Turing machine concept as a reason for seeking new principles of computing; the apparent conflict between the idea of ‘computing as compression’ and the fact that computers may create redundancy - and how the contradiction may be resolved; monotonicity and non-monotonicity of functions; information theory as a basis for ‘computing as compression’; computer models of a proposed ‘new generation’ computing system dedicated to information compression by pattern matching, unification and metrics-guided search (and how, within this framework, the effect of re-write rules may be imitated; how information may be transposed from one place to another; and how the effect of procedural programming may be achieved); computational complexity of information compression; the relationship of current proposals to research on inductive inference and algorithmic information theory.

1 Introduction

In a previous article in this journal [25], I discussed the proposition that "the storage and processing of information in computers and in brains may often be understood as *information compression*." (p 107). In a later article [15], two of my colleagues offer a critique of the computing aspects of [25] and research on the more specific conjecture that all kinds of computing and formal reasoning may usefully be understood as information compression.

The present article answers the main points in [15], tries to correct the many inaccuracies and misconceptions in that article, and discusses related issues. It is intended that this article should be comprehensible without recourse to the previous articles.

Information compression may be seen as a process which increases the *simplicity* of data (by reduction of redundancy) whilst preserving as much as possible of the non-redundant descriptive *power*. Hence the mnemonic 'SP' applied to ideas in this area. Since information compression is a rather broad concept, 'SP' is intended as a label for the relatively specific proposals described at the beginning of Section 2, below, and elsewhere [28]

1.1 The Development of Theory

Within [15] there seems to be an underlying misconception about the nature of theories and how they arise. The authors appear to believe that a theory springs into life, fully formed with all its details in place. But a theory normally starts as the germ of an idea which must then be developed until the theory can provide a coherent account of the phenomena which it seeks to address. This process of development is often compared aptly with the process of solving a jigsaw puzzle, except that the development of theory often takes years and can require significant backtracking when false leads have been followed.

The SP theory is based on an idea whose tentative status is clearly marked by the use of the word 'conjecture'. The jigsaw puzzle has reached the stage where some of the pieces have been put into groups and a few have been tentatively fitted together.

1.2 Motivation for Research

This misconception about theories and how they are developed manifests itself most clearly in observations in [15] about what the SP theory supposedly 'claims'. Any good research programme should be guided by some kind of vision of where it may lead and the potential benefits in terms of science or applications or both. Without this vision, much of the motivation for the research is missing. There is no conflict between this idea and the undoubted fact that scientific advances can arise from chance occurrences and unplanned insights. Without the longer view, we would have no way of recognising the significance of chance discoveries when they occur.

The future possibilities and practical benefits that I see for this research programme have been described most fully in [23], Chapter 6, and more briefly elsewhere. In the broadest terms, the two possibilities which provide the main motivation for this research programme are:

- Integration and simplification of concepts in computing and cognition.
- The development of a 'new generation' computing system with potential advantages over conventional computers.

It is quite inaccurate to describe these and related possibilities as ‘claims’ for the SP theory [15]. It is obvious that they are sketches of where the research may lead which are necessarily tentative. If they were totally predictable then there would be no research to be done. Their importance, as I have said, is to provide the motivation for the research.

2 The SP Conjectures

Section 4 of [15] considers some aspects of the conjectures on which the SP programme is based. In summary, these conjectures are:

- The *biological conjecture* that the storage and processing of information in brains and nervous systems may often be understood as information compression. Aspects of this conjecture have been discussed mainly in [25] and [23], Chapters 2 and 3. Although it is currently out of fashion, this idea has a good pedigree and may even be regarded as established fact.
- The *computational conjecture* that (in artificial systems) *all kinds of computing and formal reasoning may usefully be seen as information compression*. A qualified version of this idea was the subject of [22]: “The organisation and use of any kind of formal system, knowledge structure or computing system may usefully be seen in terms of the management of redundancy in information.” (pp. 518-519). The phrase “management of redundancy” was used rather than “compression” because of uncertainty at that stage about how the creation of redundancy by computers might be explained (see Section 2.3, below). In subsequent publications (e.g., [23]) the conjecture has been strengthened to essentially its present form.
- Part of the computational conjecture as it is being developed in this programme of research is the *compression conjecture*: the idea that information compression to support all forms of computing and formal reasoning may be achieved by the comparison or matching of patterns, the merging or ‘unification’[1] of patterns which are the same, combined with some kind of metrics-guided search (hill-climbing, beam search etc.) to find as much redundancy as possible for a given amount of processing ([22], Sections 2.6 and 2.7; [23], pp. 136-139, [28]).[2] An implication is that *all* kinds of information compression may be understood in these terms ([25], p. 109).

The computational conjecture is the main focus of attention in [15].

2.1 Common Sense

One can sympathise when the authors of [15] characterise the computational conjecture as

1. *Unification* in this research programme means a simple merging of identical patterns, close to the non-technical meaning of the word. Although there is some risk of confusion with the more complicated concept of ‘unification’ in logic, the term has been adopted because no other word captures the intended meaning so well.
2. In the rest of this article, pattern matching, unification and metrics-guided search will be abbreviated as PMUS.

"a stance that seems almost to defy common sense" (p. 214) because, in my experience, other people react in a similar way. But common sense is a fickle mistress in science. If we were to follow it slavishly we would still believe that the world is flat and that the sun moves round the earth. And the counter-intuitive insights of relativity and quantum mechanics would never have won acceptance.

2.2 Turing Machine

> "It is not at all clear what could be gained from such a view over existing, widely accepted models of computation, such as the Turing machine." ([15], p. 214).

It is a common misconception that alternative theories in one domain must necessarily compete. Each theory may be 'correct' in its own way. Alternative theories may eventually be integrated (witness the wave theory and the corpuscular theory of light).

Turing's achievements are, of course, monumental. But the concept of a Turing machine, while it provides many insights, also leaves many questions unanswered. Since Turing's time there has been much research in areas like logic progamming, functional programming, object-oriented design, expert systems, machine learning, natural language processing and many others. There is a pressing need to integrate and simplify concepts which have been developed in areas like these. Information compression may provide that unifying framework, either in conjunction with the Turing model or, perhaps, independently of it.

2.3 A Contradiction

The next point in [15] (p. 214) is potentially lethal for the proposition that computing may be understood as information compression: "... one wonders how such a theory could be reconciled with the obvious counter example - the set of algorithms whose express purpose involves an increase in the redundancy of a set of data." ([15], p. 214). Here is an example of a simple function which has this effect:

```
silly()
{
        print('A') ;
        silly() ;
}
```

The creation of redundancy in this way conflicts with the idea of 'computing as compression' because compression requires redundancy to be decreased.

Since this is such an important point, it surprising that it is given so little prominence in [15]. It is not clear why the authors have omitted to describe an explanation of the paradox which appears in the very source which they have cited [23]:

> "The answer seems to lie in the idea that the search process may take more than one path through the search space and may offer more than one solution to any problem. In general terms, an SP system can, in effect, create redundancy by showing a variety of different 'good' answers to the unification search. Here is an example, using the SP notation ..., to show how this might happen:

[([N _][V _][ADV _])
[N john]
[N mary]
[V walks]
[V runs]
[ADV fast]
[ADV slowly]]

=>

[([N john][V walks][ADV fast])
[N mary]
[V runs]
[ADV slowly]]

OR

[([N mary][V walks][ADV fast])
[N john]
[V runs]
[ADV slowly]]

OR

[([N john][V runs][ADV fast])
[N mary]
[V walks]
[ADV slowly]]

etc." (pp. 141-142).

For readers unfamiliar with the SP notation, '(...)' represents a sequence, '[...]' represents an unordered collection in which items may be repeated (the same as a 'bag' in logic), and '_' is a place-marker for missing information, rather like an unnamed variable in Prolog. A 'symbol' (which is an 'atom' of information in the notation) is any string of characters bounded by spaces or brackets. The arrow ('=>') is not part of the notation - it merely marks the transformation (by compression) from one structure to another. The distinction between upper- and lower-case letters has no formal significance.

Each of the three alternatives in this example illustrates a simple idea which lies at the heart of the SP programme: that redundancy may be reduced by the merging or unification of identical patterns. In each case, two of the three instances of 'N' have been merged, and likewise for 'V' and 'ADV'. The effect of these unifications is to 'pull' structures like [N john] and [V walks] into position within the ([N _][V _][ADV _]) structure.

The effect of a search process which can find *alternative* unifications is to create alternative three-word 'sentences' which, collectively, contain significant amounts of redundancy. The process may work 'backwards': a collection of 'sentences' like these may be compressed to produce a 'grammar' like the one shown at the top of the example above (see [23], pp. 148-150).

2.3.1 Recursion

Here is another example to show how unification of patterns can mean the creation of redundancy. Consider this simple structure in SP notation:

[Loop (Step1 Step2 [Loop _])].

The variable ('_') in this structure represents an arbitrary amount of missing information so that [Loop _] should unify with [Loop (Step1 Step2 [Loop _])] giving [Loop (Step1 Step2 [Loop _])] as the result.

Clearly, this process is recursive and may continue without limit. On each cycle, there is a unification of patterns in accordance with the principle of compression. But because the structure is recursive, there is no overall reduction of the size of the structure at the end of each cycle. More importantly in the present context, an 'audit trail' or 'trace' of the *behaviour* of the program will show large amounts of redundancy.

2.4 Monotonicity and Non-Monotonicity

> "... in [9] it is proven that non-monotonicity is required as an element for a universal basis of computation. In this light it is odd that Wolff attempts to model with SP the entire class of both monotonic and non-monotonic computable functions using only the class of strictly monotonic compression functions." (p. 214).

In the article referred to, Minsky [9] argues that, in a network of 'elements' of certain kinds, organised in a particular way, at least one of the elements must be non-monotonic in order for the network as a whole to be non-monotonic. The point is not actually "proven" (it depends on the words "it is easy to show that ...") but it is plausible enough. However, the arguments have been applied only to that particular type of system and may not be applicable to other models.

In [9], a 'response function' is *monotonic* if, for any two sets of input values where one set is a sub-set of the other, the same relation holds for the corresponding two sets of output values. The function is *non-monotonic* if the relation does not hold. Examples are shown in Figure 1.

Monotonic

Input	Output
{A, B}	{X, Y}
{A, B, C}	{X, Y, Z}

Non-monotonic

Input	Output
{A, B}	{X, Y, Z}
{A, B, C}	{X, Y}

Figure 1. Inputs and outputs for a monotonic function and a non-monotonic function. Following the usual convention, sets have been marked with '{' and '}'.

Compression in an SP system would normally be monotonic, assuming that the effectiveness of the compression process does not vary or that the proportion of non-redundant information which is discarded (if any) does not vary. With 'lossless' compression, the limiting case is where two inputs of different sizes have the same amount of non-redundant information and may therefore be compressed to the same size.

Although an SP system may itself perform compression which is monotonic, this does not mean that functions executed by the system necessarily have that property. This is because the meaning of the terms ‘input’ and ‘output’ are different in the two cases. This may be seen from the outline description, below, of how SP functions may be executed (Section 5.5) and the much fuller description which is given in [26].

In brief, the argument is that:

- In simple cases, a function may be represented as a set of patterns of I/O values and the effect of ‘table lookup’ may be achieved by pattern matching, unification and search.
- In more complex cases, a function may be represented as a compressed set of patterns but output values may still be obtained by PMUS.

In each of the two examples shown in Figure 1, it is not hard to see that one of the two rows may be discriminated by the matching of an ‘input’ pattern with patterns in the ‘input’ column so that the corresponding ‘output’ may be delivered. This process is independent of the ‘output’ and is thus indifferent to the distinction between monotonic and non-monotonic functions.

3 Information Theory

My earlier article [25] traces a connection between Shannon’s concept of redundancy [15] and the concept of redundancy as relatively frequent repetition of patterns. From this, it is argued that redundancy in information may be reduced by decreasing the repetition of those patterns which repeat more often than other patterns of the same size. Relatively frequent repetition of patterns may be detected by searching for matching patterns. Possible connections with algorithmic information theory (e.g., [1]) are briefly described.

If the authors of [15] had wished to demonstrate “a confusing terminological hodgepodge, fraught with ambiguity, inconsistency, and imprecision” (p. 214), one would expect them to have made their case from the discussion in [25]. Since these qualities are merely asserted rather than demonstrated, no other response is required.

4 SP Models

In Sections 6 and 7 of [15], existing prototypes of the SP system, and their capabilities, are examined in general terms. Before offering responses to observations made in those two sections, I will describe current models in outline.

4.1 SP6

SP6 is an early prototype described in [31], [23] (Chapter 5) and [24]. Examples from this prototype are presented in [26] in support of the idea that the design of functions, and their execution, may both be achieved by information compression.

The main aim in developing this prototype was to explore whether and how it might be possible to integrate such operations as unsupervised inductive learning, logical deduction, probabilistic inference, information retrieval, the execution of functions, pattern recognition and planning within a framework of information compression by pattern matching, unification and search. Examples from the model are presented in the sources cited above.

The focus has been on *integration* within one uniform framework: “I stress that *the variety of capabilities illustrated in this section are the product of one unifying concept - the search for efficiency in information.*[3] Any one of the functions described can be done more effectively by some existing computing system or program but, as far as I know, there is no existing system which embraces all these capabilities as a product of one organizing principle.” ([23], p. 147, emphasis in the original).

Amongst the several shortcomings of SP6 there are three main weaknesses: the search method is inefficient and, more importantly, will not scale up to handle realistically large volumes of data; it only delivers one ’best’ answer to problems instead of the alternative answers which are often required; and it has only a very restricted ability to discover patterns which are ‘discontinuous’ or ‘fragmented’.

4.2 SP20 and SP21

The main goal in developing SP20 has been to overcome those three main weaknesses of SP6 whilst maintaining an ability to integrate diverse functions ‘cleanly’ within a framework of information compression. A subsidiary goal has been to achieve these ends with a model which would lend itself to parallel processing.

As described in [29], the search method in SP20 is *much* more efficient than in SP6 and appears to be scaleable to large bodies of information. In a serial processing environment, the time complexity of the model is $O(N^2)$and the space complexity of the model is O(N), where N is the number of symbols in the information supplied to the model. Analysis of the model suggests that a parallel version (not yet attempted) would have the same space complexity and a time complexity approaching O(N), depending how well the parallelism is applied. The model has a robust ability to find recurrent patterns which are discontinuous or fragmented (i.e., subsequences) as well as coherent substrings. It can deliver a graded series of alternative matchings for patterns but there is more work to be done on the formation of alternative unifications.

SP20 is a substantial advance over SP6 but in some respects the earlier model is better. Both models are quite restricted in what they can do and there are several significant problems still to be solved. These problems are described in [29].

In terms of potential practical applications, SP20 is strongest in the area of information retrieval. With a view to possible future ‘spin-off’, SP21 is a variant of SP20 which is dedicated to ‘best-match’ information retrieval [27]. It incorporates some refinements of the search method in SP20 and of the metric used to guide the search.

---

3. *Efficiency* is the ratio of non-redundant information to total information. This ratio is increased by the extraction of redundancy from information and the consequent compression of the information (provided any accompanying loss of non-redundant information is not too great).

5 Comments

5.1 The Development of Computer Models

It is suggested in [15] that the development of working models has been at the expense of “seriously addressing important theoretical problems” (p. 215). This observation appears to derive from the view that theoretical work should be done exclusively by ‘pure thought’. This was necessarily the case before computers were invented. But since that time, many people have discovered that the process of developing computer models and running them is a powerful aid to theorising. The technique is now widely used in many areas including mathematics (e.g., [1]).

There is, of course, a risk of blind ‘hacking’. But if computer models are developed in a disciplined way in conjunction with hard thinking about the theory which they are designed to express, the process of creating a model can sharpen one’s view of theoretical problems. And the process of running the model on appropriate examples can quickly reveal weaknesses in one’s understanding that might otherwise never come to light.

Even with the aid of computer modelling, this kind of theory development is hard. There are many opportunities for false moves and many blind alleys that can be entered. The goal at all times is to preserve simplicity in the concepts and, at the same time, to accommodate as wide a range of phenomena as possible (‘simplicity’ with ‘power’!). The temptation to add *ad hoc* mechanisms to meet specific problems must be resisted.

5.1.1 Rigorous Definition

Regarding the suggestion in [15] that the meaning of the ‘wild card’ object (i.e., the variable (‘_’)) should, by now, have been “rigorously” defined, the exact nature of this construct within the SP system, or whether it is needed at all, has indeed been a recurring puzzle. Of course, an arbitrary formal meaning could have been given to this construct at an early stage but this would almost certainly *not* have been the route to a good theory. Concepts within a developing theory need to be bootstrapped with care to ensure that the overall structure is sound.

My tentative view has been that the need for the variable might disappear when search methods were improved. But until an improved search method had been devised (with the aid of computer modelling), it was not easy to resolve this point. It turns out that, at least in applications like best-match information retrieval, the search method in SP20 and SP21 completely eliminates the need for any wild card object.

5.2 What Are the Models Doing?

It is correctly observed in [15] that a judgement about what the models are doing depends on how the output is interpreted. This, of course, is true of any computer system. Also relevant is our knowledge of the internal workings of the given system. We have few qualms about the conclusion that a pocket calculator is ‘doing’ arithmetic or that a Prolog system is ‘doing’ logic. But these judgements rest ultimately on the patterns we see on the LCD or computer screen, our knowledge of how the systems work, and how these things relate to our concepts. Making these connections is a process of perception and analogy.

I believe it is reasonable to interpret the output and the workings of current SP models in terms of such concepts as information retrieval, unsupervised inductive learning, deduction, abduction and others. But making such judgements is not in any way a claim that the models perform these functions comprehensively or perfectly. It was reasonable for the Wright brothers to claim that they had demonstrated powered flight, even though they were not able to carry a Jumbo jet load of passengers at high speed across the Atlantic.

That current models have weaknesses has been made abundantly clear: “Any one of the functions described can be done more effectively by some existing computing system or program ....” ([23], p. 147); “OAOs have been used [in this example] rather than the more appropriate UAOs because SP6 lacks the ability to look for patterns and sub-groups within UAOs. Future versions of SP will plug this gap.” ([31], p. 272); “... this version of SP6 has actually failed to preserve one of the premises in this example (that ‘All men are mortal’) although, in designing the system, we intended that it should not lose power in its knowledge structures. We expect this weakness in SP6 to be cured in later versions of SP ...” ([31], p. 273). “Notice that this example would not give a valid conclusion if the relationship were, say, ‘not-equals’. This and other examples point to the conclusion that negation will require special treatment in SP, ...” ([31], p. 152); With more space, many other similar quotations could be given.

“SP can imitate the effect of a re-write system”. This quotation from [31] is presented in [15] as an example of a proposition with little support. The quotation is actually a fragment of a sentence which appears in a section on ‘Related Work’: “Although SP can imitate the effect of a re-write system it does *not* use any re-writing technique in the technical sense of that term ...” (p. 266, emphasis in the original). From the context and from the fuller sentence it can be seen that the aim at that point was to differentiate SP concepts from re-writing systems rather than justify the first proposition in the sentence. Relevant evidence (which could have been quoted if the first proposition had been the focus of attention) was presented in [22], Sections 5.1 and 5.4. The SP6 model (which was the subject of [31]) can imitate re-writing in a restricted way, as discussed below (Section 5.3).

“The SP prototype ... demonstrates an ability to make logical inferences which is comparable with the capability of systems like Prolog”. This quotation in [15] from an unpublished document ([30], p. 18) certainly gives a misleading impression of what the SP prototype could do if “is comparable with” is taken to mean “is as good as” rather than the stricter meaning (which was intended) of “may be compared with”. The impression that any undue claim for the prototype is being made is less strong if the immediately following sentence is given: “*There are still problems to be solved* but we expect the *mature* SP system to be able to store logical propositions and to make logical inferences at least as well as existing systems for theorem proving and logic programming.” (emphasis added). And the impression is even less strong when published accounts of SP prototypes point out their shortcomings so clearly (see above). It does not require “close examination” to discover that a negation operator is missing and is probably required because this has been clearly stated and discussed ([31], pp. 274-275).

5.3 Re-write Rules

In their simplest form, re-write rules look like this:

S -> N V ADV
N -> John
N -> Mary
V -> walks
V -> runs
ADV -> fast
ADV -> slowly.

With this example (similar to the example shown in Section 2.3), a 'sentence' may be created by re-writing S as N followed by V followed by ADV (i.e., replacing S by those three symbols, in order), then re-writing N by 'John' or 'Mary', re-writing V by 'walks' or 'runs' and re-writing ADV by 'fast' or 'slowly'. In more realistic examples, re-writing is done through three or more 'levels'. In some applications, rules of this type may be run 'backwards'.

In the SP framework, each rule may be seen as a sequential pattern which may be represented in the SP notation like this:

[((N _)(V _)(ADV _))
(N John)
(N Mary)
(V walks)
(V runs)
(ADV fast)
(ADV slowly)].

There are some differences in style between this example and the one given in Section 2.3 but it would take us too far afield to discuss them and they are not relevant to the point in hand. When this structure is processed by SP6, it is compressed into this form:

[((**N** John)(**V** walks)(**ADV** fast))
(N Mary)
(V runs)
(ADV slowly)].

Bold type is used to mark symbols which are the result of unification. The effect of pattern matching and unification in the example has been to 'pull' the elements (N John), (V walks) and (ADV fast) into positions within the sentence framework thus creating a simple sentence. SP6 has only a very restricted ability to achieve this kind of effect through two or more 'levels' (see [23], p. 158) and, as indicated earlier, it only finds one 'best' set of unifications and cannot, therefore, create alternative sentences.

As illustrated by this example, re-writing may be seen as a process of information retrieval where the retrieval 'key' is the left side of the rule and the unmatched symbol or symbols in the retrieved pattern correspond to the right side of the rule. Likewise, re-writing (and retrieval by key) may be seen as the 'de-referencing' of an identifier, where the left side of the rule is the identifier and the right.side is the identified object.

This kind of information retrieval by key or de-referencing of an identifier may be seen as a simple case of a more general idea: that any sub-sequence of a sequential pattern may be used to retrieve that pattern (perhaps also with other patterns in which the same sub-sequence occurs). And this in turn may be seen as a relatively simple case of 'best-match' retrieval in which there may be mis-matches between the query and the retrieved pattern. Best-match retrieval is discussed much more fully in [27].

5.4 Transposition of Information

As described in [15], a Post canonical system [11] is one "comprising a single axiom together with productions of the form: g$ -> $h, where g and h are constants and $ is a 'wild card' operator which can match zero or more symbols." (p. 216). The key point is that, with a system like this, it is possible to transpose information from one part of a sequence to another. If, as appears to be the case, "Post canonical systems constitute another universal model of computation and ... *all* formal systems may be expressed as Post canonical systems." ([15], p. 216, emphasis in the original) and if, as may be the case, transposition cannot be accommodated within the SP framework, then there is a problem for the SP computational conjecture. There would be a problem even if Post canonical systems were not 'universal' because they clearly support a form of 'computing' or 'formal reasoning'. There is in any case a very reasonable expectation that any general-purpose computing system should be able to transpose information from one location to another.

That an SP system (or any other computing system) should be able to transpose information does not, in itself, mean that that capability has to be provided as a *primitive* element of the system. Although it cannot yet be demonstrated in any SP model (and has, indeed, not been a primary focus of attention in work done to date), it appears that transposition of information may be achieved as a by-product of the processes of pattern matching, unification and search which have been identified tentatively as key mechanisms for information compression.

Consider an example like this:

```
[((Subj _)(V1 _)(Obj _))
((Obj _) is (V2 _) by (Subj _))
(Subj John)
(V1 eats)
(V2 eaten)
(Obj food)].
```

In the light of the discussion and examples in Section 2.3 and the first part of Section 5, it should be clear that an SP system that is capable of creating alternative unifications should create two alternative 'best' versions of this structure:

[((**Subj** John)(**V1** eats)(**Obj** food))
((Obj _) is (V2 _) by (Subj _))
(V2 eaten)]

and

[((Subj _)(V1 _)(Obj _))
((**Obj** food) is (**V2** eaten) by (**Subj** John))
(V1 eats)].

It is a relatively small step to see how an 'active' sentence may first be 'parsed' and then reconstructed in the 'passive' form with its 'subject' and its 'object' transposed:

```
[(John eats food)
((Subj _)(V1 _)(Obj _))
((Obj _) is (V2 _) by (Subj _))
(Subj John)
(V1 eats)
(V2 eaten)
(Obj food)]

=>

[((Subj _)(V1 _)(Obj _))
((Obj _) is (V2 _) by (Subj _))
(Subj John)
(V1 eats)
(V2 eaten)
(Obj food)]

=>

[((Subj _)(V1 _)(Obj _))
((Obj food) is (V2 eaten) by (Subj John))
(V1 eats)].
```

In this example, the order of the words in the original sentence has been lost but could, perhaps, have been preserved by attaching appropriate tags to the ((Subj _)(V1 _)(Obj _)) pattern in the 'grammar'. A more comprehensive grammar would, no doubt, mark the 'semantic' connections between active and passive forms and between 'eats' and 'eaten'.

The general point here is that an ability to transpose information is not necessarily primitive. It may be an 'emergent' property of a system which is dedicated to PMUS. If it can be shown that Post canonical systems can be imitated within the PMUS framework of an SP model and if it is accepted that Post canonical systems are 'universal' then the SP framework is universal too.

5.5 Procedural Programming

5.5.1 A Simple Language

> "[The SP language] is a language with no reserved words; no defined data types; virtually no data structures; no arithmetic/logical operators; no input/output primitives; no iterative constructs; and, perhaps most importantly, no means of controlling program flow. In fact, it is difficult even to conceive of how an algorithm could be represented, let alone executed, in the SP language." ([15], p. 216).

The SP language has been correctly characterised here as lacking in many of the features which are commonly provided in conventional systems. And, with the possible exception of a negation operator, there is no intention of adding features like these in the future as primitive components of the language.

The parsimonious design is motivated by the need for parsimony in the SP theory. The aim of the research programme is to see whether or not it is possible for ‘conventional’ concepts to be understood as *emergent* properties of a system built from PMUS, and, if so, how. If the kinds of features mentioned were simply bolted on to the SP language then the conceptual framework would slide back into the conventional mould and the possibility of achieving a new and improved synthesis of ideas would quickly recede.

5.5.2 The Execution of Functions by Information Compression

The way in which software might be designed *and executed* by information compression has been discussed quite fully in [26]. The gist of the argument is this:

- Conceptually, any function is a set of patterns which maps inputs to corresponding outputs. This is the ‘extensional’ view of functions which is widely recognised in computer science (see, for example, [16], p. 279).
- In simple cases (e.g., a thermostat function or the ‘exclusive OR’ (XOR) function), a function may be defined by an unmodified set of I/O patterns. But in most cases, this kind of representation would be too large to be practical. Consequently, the set of patterns must be compressed.
- The devices commonly used for representing the structure of a computer program (named functions with or without arguments; *while* loops; *fo*r loops; recursion; class hierarchies with inheritance; etc.) may be seen as examples of devices commonly used for information compression (‘chunking with tags’; ‘run-length coding’; ‘schema-plus-correction’). Thus, a set of patterns which has been compressed by means of these devices will have a form which resembles the organisation of a conventional computer program. This is an ‘intensional’ definition of the function.
- If a compressed set of patterns contains one or more recursive structures, then that compressed structure may represent an *infinite* range of I/O values (cf. Section 2.3, above). Recursion may be seen as an example of the run-length coding device for information compression.
- As discussed in [26], the *execution* of a function may also be achieved by information compression:
    - If the function is represented by an explicit (uncompressed) set of patterns, then the matching and unification of an ‘input’ pattern with (the ‘input’ side of) one or more entries in the set of patterns achieves the effect of ‘table lookup’. The ‘output’ in this case is the unmatched symbols in the entry or entries which have been accessed. This is similar to information retrieval by the partial matching of patterns (cf. Section 5.3, above).
    - Even if the function is a *compressed* representation of I/O patterns, ‘output’ values may be accessed in a similar way: by matching and unification of ‘input’ values with patterns within the compressed structure, and the retrieval of the unmatched symbols

> which are associated with the resulting unifications. Examples of how this may be done are given in [26]. The examples and discussion in Sections 2.3 and 5.3, above, are also relevant.

This outline, and the fuller account in [26], provides a basis for answering points raised in [15]. The three main points are considered in the next three sub-sections.

5.5.3 Composition of Functions

> "... SP is unable to handle compositions of functions ...." ([15], p. 216).

Taking "compositions of functions" to mean the 'calling' of one function by another, this seems to correspond to recursive embedding of 'tagged' structures which has been demonstrated by SP6 ([23], p. 149; [26], Section 4.1). However, for essentially the same reasons that current SP models have no more than a very restricted ability to de-reference identifiers through multiple levels (see Section 5.3, above), they cannot imitate the familiar 'cascading' of function calls. This appears to be a defect in current models rather than a more general defect in the framework of ideas.

5.5.4 Explicit I/O Values?

> "... the idea of having to store explicitly tabulated function values for all the possible inputs for every function that may be required hardly seems a practicable proposition." ([15], p. 216).

Of course, explicit values would only be used for small sets of I/O patterns. In all other cases, as explained above, compressed representations would be used.

5.5.5 'Algorithmic' Computation

> "... the proposed approach relies on the relevant values somehow being available in advance. ... in SP there is no possibility of being able to compute a function value algorithmically. This means that an SP system would in fact require some alternative means - a program written in a 'conventional' programming language running on an 'ordinary' computer. ... The failure of SP to handle functions adds further weight to the argument that programming in SP is not possible. ... an inability to implement functions suggests an inability to execute programs." ([15], p. 216).

These quotations show that the thrust of the computational conjecture has not been understood by the authors of [15]. It is part of the computational conjecture that computing the value of a function 'algorithmically' and the execution of 'programs' may be achieved by information compression. I believe the arguments summarised above (Section 5.5.2) and presented more fully in [26], and the examples in Sections 2.3 and 5.3 above provide good support for the hypothesis.

5.5.6 'Broad' and 'Narrow' Searching

Before leaving the topic of procedural programming, it is pertinent to mention the distinction between 'broad' and 'narrow' searching in SP systems:

> "Within the scope of the hill climbing concept, there is another dimension:

1 *“Broad searching* means searching which is relatively thorough. The cost of a thorough search is that it is hungry for computing resources and is likely to need high levels of parallelism in processing to achieve adequate speed. The benefit of thorough search in a computer system is that the system can be relatively flexible and intelligent.

2 *“Narrow searching*. Where computational resources are limited - as they are in ordinary computers - the search process may be constrained in various ways so that relatively large portions of the search space can never be reached. This kind of constraint reduces computational costs but means that the computing system has less flexibility and more of that ‘brittleness’ and stupidity which characterises current systems.

“Amongst existing computer languages, those classified as ‘declarative’ seem to provide a relatively broad search compared with those classified as ‘procedural’. Procedural languages seem to have been developed as a response, at least in part, to the limited power of present-day computers. The greater power of computers in the future should allow the declarative style to flower.” ([23], p. 139; see also [22], p. 526).

6 Computational Complexity

Section 8 of [15], which deals with questions of computational complexity, contains more than its fair share of errors and misconceptions. I will take each one in turn.

6.1 Complexity Ignored?

“... an issue of paramount importance to the practicability of the proposed scheme, and one that until very recently has been completely ignored in the SP theory, is the question of the computational complexity of the SP compression process.” (p. 216).

Far from having been “until very recently ... completely ignored”, the question was clearly recognised in the programme of research on unsupervised inductive learning from which the computational conjecture arose:

“One feature of this program [MK10] (and SNPR) which deserves special attention is the way the combinatorial explosion is handled. For most values of *s* there is a huge number of possible dictionaries[4]. The number of alternative groupings that have to be considered [by the program] at any one time is constrained by the fact that only pairs of neighbouring elements are recorded. This restriction does not prevent the program from building arbitrarily large elements by successive pairings. The number of possible pairs which have to be recorded at any one time is never more than n - 1 (where n is the number of characters in the sample) and is often less than this. There is never any risk of being swamped by the number of possible groupings. The technique can be characterised as a hill-climbing search of the space of possible dictionaries.” ([21], p. 169).

What the authors of [15] seem not to understand is that, because most AI problems have astronomically large problem spaces, principles of metrics-guided search (hill climbing, beam

4. *s* is the size of the set of patterns to be discovered by the program.

search etc.) are taught in every AI textbook and are ‘common currency’ for researchers in AI. It is widely recognised that, while exhaustive search is normally impossible, hill climbing and related techniques can often produce useful results without undue computational demands. It is also widely recognised that, because these techniques save computation by ignoring parts of the search space (usually very large parts), they cannot guarantee that the best possible result will be found.

From the beginning of the current programme of research into the computational conjecture, these principles have been clearly flagged:

> “There is no algorithmic way of ensuring that C is always maximised.[5] Except for trivially simple cases, it is always necessary to use the kinds of *search* technique familiar in artificial intelligence (most notably, *hill climbing*). These techniques cannot guarantee a perfect solution, but they provide a practical means of finding at least an approximation to the desired goal.” ([22], p. 522, emphasis in the original).

and

> “Finding a good set of unifications for the repeating patterns in a body of information amongst all the many possible sets of unifications means a process of *searching*. Because the search space is usually very large, often infinite, it is necessary to use a *hill-climbing* technique - or something equivalent - to achieve an effective search in a reasonable time.” ([31], p. 265, emphasis in the original).

6.2 Early SP Models

> “Despite token recognition in the SP literature of the impracticability of attempting exhaustive searches in the input corpora, it is nonetheless this very activity upon which the SP prototypes have been based. For example, the system most often cited in the SP literature, namely SP6, attempts an exhaustive substring search and is consequently of no practical use for all but the most trivial inputs.” ([15], pp. 216-217).

The inefficiency of the search process in early SP models is not a secret and has been clearly stated in published descriptions, e.g.: “A relatively unsophisticated kind of hill climbing technique is being used [in SP6]. In future versions of SP, there is scope for substantial improvements in the efficiency of the searching process (using ‘efficiency’ here in the sense of cutting out unnecessary repetition of operations).” ([23], p. 145).

Very much more efficient and scaleable methods of finding repeating substrings were available at the time SP6 and other early SP models were developed. For example, my own MK10 program uses a hill-climbing technique which enables it to perform very effective searches of realistically large quantities of unsegmented natural language text with quite modest computational resources (see, for example, [19]).

The reason that one of these more efficient and scaleable methods was not used in SP6 (and other early models) was that the primary purpose of developing these models was to understand more fully how diverse computing functions might be *integrated*. The simple and very inefficient

5. C is a value whose formula is given in [22] which reflects that part of the redundancy in a body of sequential information which is due to repeated substrings.

search method that was used was merely a stop-gap while the integration issue was explored. Using the earlier method might have led to 'tramlining' in one's thinking, obscuring the kind of reorganisation needed to achieve integration. It was reasonably clear at those early stages (and even more clear now) that the organisation of the MK10 model was not appropriate for an SP model.

Given previous experience with this kind of search, and given the general understanding of how metrics-guided search can tame the biggest search problems (Section 6.1), there was no doubt that better search methods with acceptable computational complexity could be introduced into SP models at a later time. This expectation has been born out in the SP20 and SP21 models, described above (Section 4.2).

6.3 Exhaustive Search

> "Having seen that exhaustive searching in SP is not a feasible proposition, it is now relevant to state that all of its claimed capabilities would appear to require this very mode of execution for their correct operation ...." ([15], p. 217).

This, of course, is nonsense. Unsupervised inductive learning and best-match information retrieval are just two of the capabilities targeted in the SP programme which can be done very effectively with non-exhaustive searching.

> "To take but one example, consider the case of procedural programming in SP. If it is not feasible to search for all the redundancy in the input, then it cannot be guaranteed than any particular pair of equivalent substrings will in fact be merged by the system. This in turn means that the particular 'compression' of the 'program' necessary for its correct operation cannot be guaranteed, which leads to serious doubts over the determinism of otherwise correct SP 'programs'." ([15], p. 217).

The logical *non sequitur* here is the belief that correct operation of a procedural program requires a guarantee that "any particular pair of equivalent substrings will ... be merged by the system". What is required (*inter alia*) for a procedural program to execute correctly is a guarantee that *some* particular pair or pairs of equivalent substrings will always be merged. The existential quantifier is needed, not the universal quantifier.

Consider, for example, the front end of a compiler for an ordinary programming language on an ordinary computer - a typical deterministic procedural program. The front end of a compiler must parse the source code and produce one 'correct' translation if the source code is correct, or one set of error messages when there are errors. (Of course, the back end of a compiler is also a deterministic program with similar characteristics.)

Parsing is an operation which is naturally non-deterministic. Grammars and languages are frequently ambiguous so that there are often two or more 'good' ways of parsing a particular text. Ambiguity is a prominent feature of natural languages and their grammars. Parsers for natural languages are normally designed to give alternative parsings where ambiguity exists.

The challenge for the designer of a compiler for a programming language is to eliminate ambiguity. The grammar of the programming language to be parsed is normally expressed in a suitable form (e.g., LL(1)) so that, with a suitable parsing technique (e.g., recursive descent), ambiguity is removed and the parser always delivers one 'best' parsing.

The overall effect of these design decisions is to constrain the process of matching patterns in the source code against patterns in the grammar so that many potential comparisons are never made. The application of this constraint has the effect of reducing the computational demands of the parser. More importantly in the present context, the constraint makes the search so simple that we can be sure that the ‘correct’ comparisons and unifications are always made.

The same principles may be applied to an SP system. If the search process is constrained (cf. ‘narrow searching’ described in Section 5.5, above), the computational demands of the system will be reduced but any ability which the system might have had to find alternative answers to problems will be progressively reduced. The system may thus be made to be deterministic and to deliver one ‘correct’ result for any given input. However, the constrained system is also likely to acquire the unattractive ‘rigidity’ of conventional computers.

6.4 Tractability

> “... it should be noted that any universal model of computation should have a negligible complexity overhead in its own operation to be of use. In the case of SP the intractable nature of its own operation would, in turn, dominate the complexity of any algorithms run by it. In effect, all algorithms implemented to run under SP would take an intractable amount of time to do so, regardless of their own inherent complexity. They would become swamped by the computational demands of the SP universal model of computation itself. And this would thus make everything run under SP essentially intractable.” ([15], p. 217).

Issues raised here may be clarified by reference to Figure 2 which suggests, in schematic form, how the organisation of a conventional computer and an SP computer may be seen.

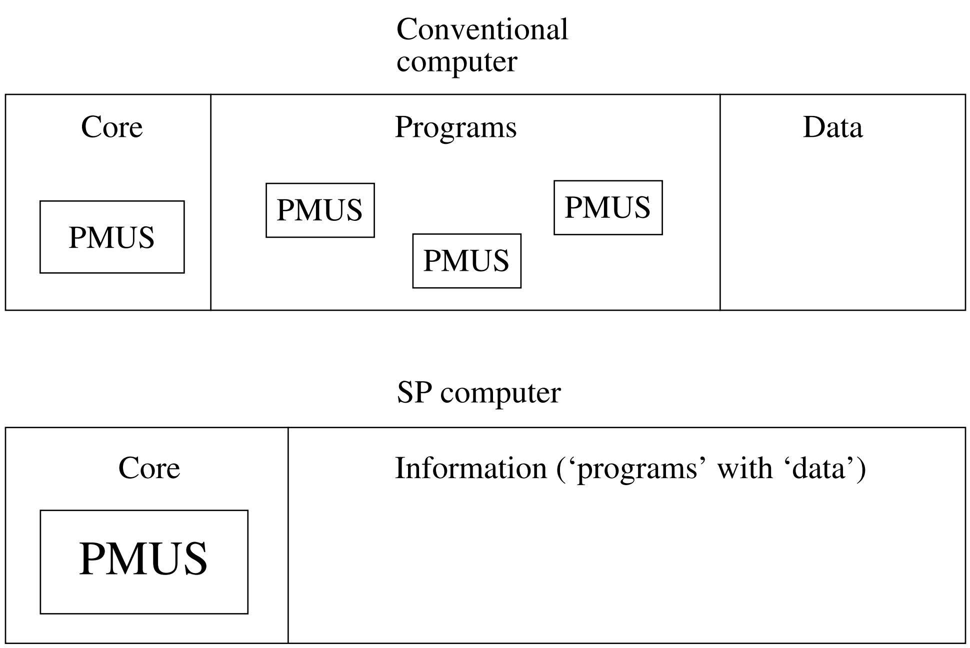

Figure 2. Schematic representation of a conventional computer and an SP computer. ‘PMUS’ means mechanisms for pattern matching, unification and metrics-guided search.

In the SP view, a conventional computer contains a ‘core’ (the CPU with mechanisms for accessing memory), part of which may be seen as PMUS mechanisms. As was discussed in [25], Section 5.6, memory access may be seen in those terms. The same may be true of the operations of an ALU. To make the core useful, there must also be programs, and programs, in turn, need data.

The PMUS mechanisms in the core of a conventional computer are very fast but heavily constrained so that only the simplest kind of searching is performed. To make up for this deficiency, programs frequently contain code which has the effect of supplementing the core mechanisms to achieve more sophisticated kinds of PMUS. This kind of augmentation of the basic PMUS mechanisms can be seen most clearly in applications like database searching and 'grep'. Less obviously, it appears in the workings of compilers, in accessing the names of data structures and functions in programs, in 'relaxation' techniques, and in other operations and applications especially in AI, as discussed in [25] and [28]. These supplementary mechanisms are frequently 're-invented' in varied forms in many different programs.

In the design of an SP system it is intended that the core should contain one general-purpose mechanism for PMUS (which may be adapted for 'broad' or 'narrow' searching). This mechanism would be more sophisticated than the mechanisms in conventional computers and, for that reason, may well be more 'costly'. However, there should be a net saving of resources because it should then be possible to eliminate supplementary PMUS mechanisms in 'programs' and, perhaps more importantly, eliminate their frequent re-creation in different applications and systems.

In an SP system, a 'program' is a body of already-compressed information and 'data' is new information (which may or may not be compressed) which is to be merged (fully or partly) with the pre-established 'program'. As described in Section 2, PMUS is seen as fundamental in *all* kinds of computing system, including conventional computers. What is envisaged for an SP system is a *re-organisation and re-distribution* of PMUS mechanisms to eliminate unnecessary structural complexity and to maximise the benefits of compression techniques.

The quotation at the beginning of this section assumes that the SP model is inherently intractable. From the discussion in earlier parts of Section 6 it should be clear that 'broad' SP should be no more power hungry than existing heuristic systems, many of which produce useful results without undue computational demands. And the 'broad' model may be constrained so that computational demands are reduced (with a corresponding loss of flexibility and 'intelligence').

The quotation also assumes that the computational demands of SP 'programs' will be added to the demands of the core system. From the discussion in this section, it should now be clear that it makes little sense to regard SP 'programs' as having any intrinsic computational complexity. For the PMUS core of an SP system, the 'programs' are, in effect, 'data'. The computational complexity of the SP core may be analysed with respect to these and other kinds of data in the normal way.

7 Related Research

In Section 9 of [15] it is suggested that systems equivalent to the proposed SP system were "thoroughly investigated about 30 years ago (e.g., [13])" and that "neglecting its central conjecture, the SP theory is wholly subsumed by the area of Kolmogorov complexity (e.g., [7][13][32][2][8])."

If the concepts are so well established, it is not clear why they should "defy common sense" ([15], p. 214) or be misunderstood in the way that they have been in [15]. If SP systems were "thoroughly investigated" 30 years ago we would certainly expect them to be now available in the shops.

7.1 Inductive Inference

One of Solomonoff's achievements (see, for example, [13]) has been to highlight the significance of information compression in the field of inductive inference (predicting the future from the past) and the related field of grammatical inference. Notions of pattern matching and the unification of patterns were the basis of 'distributional analysis' as a method of grammatical inference in the field of 'structural' linguistics (e.g., [5]) but Solomonoff brought information compression more sharply into focus.

With pleasing modesty, Solomonoff described his own proposals as "rudimentary" ([13], p. 11), thereby acknowledging that there were many questions still to be answered. Some of these questions have been addressed in subsequent research (e.g., [4][10][18][20]) but much remains to be done. We are, for example, still a long way from achieving unsupervised inductive learning of a grammar for any natural language.

In later research (e.g., [14]), Solomonoff has proposed a general problem solving method based on algorithmic probability theory. This research differs from the SP programme in its objectives and approach. But even without these differences, there was and is much work to be done before it could be said that SP systems had been "thoroughly investigated."

7.2 Algorithmic Information Theory

> "... it seems that, *neglecting its central conjecture*, the SP theory is wholly subsumed by the area of Kolmogorov complexity ..." ([15], p. 217, emphasis added).

It is not clear how one theory may be subsumed by another if, at the same time, it is necessary to neglect its central conjecture! Although nothing is said, one may suppose that the authors of [15] have included this extraordinary qualification because they recognise that there is a basic conflict between the two theories.

A central idea in Kolmogorov complexity theory, which is perhaps better known as Algorithmic Information Theory (AIT), is that randomness may be defined in terms of the compressibility of information: "A series of numbers is random if the smallest algorithm capable of specifying it to a computer has about the same number of bits of information as the series itself." ([2], p. 48). If an algorithm can be found which is significantly smaller than the given series of numbers then the series is not random.

This idea, that an algorithm (or computer program) may generate a series of numbers or other information which is bigger than the algorithm itself is closely related to the point which was raised in [15] as a difficulty for the SP theory: "... one wonders how such a theory could be reconciled with the obvious counter example - the set of algorithms whose express purpose involves an increase in the redundancy of a set of data." (p. 214). As was discussed in Section 2.3, above, there is a possible resolution of the conflict, but this was not apparent to the authors of [15]. More to the point in this context is that, to my knowledge, AIT provides no means of explaining how 'computing as compression' might be reconciled with the algorithmic creation of a string of symbols which is longer than the algorithm itself. Consequently, the SP theory cannot be subsumed by AIT as it stands now.

What, then, is the relationship between the SP theory and AIT?

Shannon's information theory was an inspiration for the SP theory but, as was noted in [25] (Section 2.3 and later), AIT provides a view of redundancy and randomness which is different from Shannon's view [12] but may be integrated with it within a framework of pattern matching unification and search:

- As described in [25], Section 2.2, Shannon's basic measure of information may be generalised so that 'sequential' redundancy may be seen as relatively frequent repetition of patterns, where a 'pattern' may be a coherent substring or a fragmented subsequence.
- If PMUS is indeed a fundamental mechanism for the detection and removal of redundancy, then the existence of relatively frequent repetition of patterns means that the information is compressible, in accordance with the AIT view of redundancy.

Thus the SP theory seems to rest naturally on the twin foundations of AIT and (a modest generalisation of) Shannon's information theory. The main differences between AIT and the SP theory are:

- In general, the main focus of interest in AIT has been on defining concepts like randomness and in exploring associated theoretical limits. Since [13], there have been links with research on inductive inference. The field is well reviewed in [8]. By contrast, the focus of interest in the (computational part of the) SP programme has been to understand whether or not all forms of computing and formal reasoning may usefully be understood as information compression by pattern matching, unification and search.
- The concept of an 'algorithm' in AIT (see, for example, [17]) seems to have little in common with ideas about how functions may be executed by compression (as described in Sections 2 and 5, above, and in [26]).
- Because of the contradiction discussed above, AIT cannot, as it stands, accommodate the idea that computing may itself be a process of information compression.

7.3 Attempting the Impossible?

> "... it should be noted that some important results arising from this field [Kolmogorov complexity] are completely ignored in SP. For example, it is known that optimal compression of arbitrary data is not computable (see, e.g., [8][32]). But, it would appear that the SP compression algorithm, as presented in the literature (e.g., [31][23]), attempts just such a task by striving to extract redundancy until there remains none left to extract." ([15], p. 217).

That optimal compression of arbitrary data is not computable has been clearly recognised in the SP programme. Readers are referred again to the two quotations in Section 6.1 ([22], p. 522 and [31], p. 265). Since one cannot be sure that all the redundancy in an arbitrary body of information has been found, one cannot be sure that all the redundancy has been extracted.

Not only has this point been clearly recognised but, contrary to the quotation at the beginning of this section, none of the SP models have attempted to extract all the redundancy in any body of information. The early models (e.g., SP6) confine their attention to redundancy which may be attributed to repeating substrings and largely ignore other kinds of redundancy such as redundancy due to repeated subsequences (where constituent symbols are not necessarily contiguous in the data) or redundancy due to mirror images (e.g., palindromes). Hence, the published descriptions which were cited refer to *extractable* redundancy ([31], p. 270; [23], p. 147)). Extractable redundancy is a very different concept from *total* redundancy which would be needed for optimal compression or for the extraction of redundancy "until there remains none left to extract".

Furthermore, the small examples used with the early SP models are not "arbitrary data". Since they are small, they can be searched in a way which is not feasible for realistically large examples. As explained in Section 6.2, the choice of simple but inefficient search methods for early models was dictated by the focus on integration of functions and the need to avoid 'tramlining' of ideas. Later models (SP20 and SP21) use much better search methods which are not exhaustive in any sense.

7.4 Most Strings are Random

> "... contrary to the thrust of Wolff's conjecture, Kolmogorov complexity theory proves that almost all finite strings are random (and hence incompressible), but cannot be proved to be so (see again, e.g., [8][32])."

That most strings in the space of all possible universes are incompressible but cannot be proved to be so is certainly interesting but has no bearing on the truth or otherwise of the SP computational conjecture. Of course, 'incompressible' here means 'incompressible unless one is prepared to lose non-redundant information'.

Without redundancy (and thus the possibility of compression), any kind of prediction by a brain or computer becomes impossible. In a totally random (incompressible) universe, an algorithm which gives one result today may give a different result tomorrow. Data stored now may not be valid in the next nano-second. Indeed, all kinds of persistent, structured entities like 'algorithm', 'brain' and 'computer' will have disappeared in the white hiss of randomness.

That redundancy is a rare commodity in the space of all possible universes only emphasises its importance. Since all kinds of prediction depend on it, we need to be able to find it. Hence the prominence of pattern matching and search in the SP theory.

8 Conclusion

In this article, I have tried to answer the points raised in [15], to correct the many errors and misunderstandings, and to discuss associated issues.

That computing and formal reasoning may be seen as information compression is not widely recognised and certainly strikes many people as an outlandish idea. A broader view of information processing, which accommodates brains and nervous systems as well as computers, brings information compression more clearly into view. As described in [25], there is a range of phenomena across this broader canvas which, taken together, suggest rather strongly that information compression may provide a unifying theme for information storage and processing in all kinds of system, both man-made and natural.

Acknowledgements

I am very grateful to Dr Tim Porter and to Dr Chris Wensley, both of the School of Mathematics, University of Wales, Bangor, for advice and discussion about aspects of mathematics. Detlef Morgenstern of Dresden has provided many useful observations on these ideas and constructive comments on earlier drafts of this article.

The preparation of this article was supported in part by a grant to Dr J G Wolff from the UK Science and Engineering Research Council for "The Further Development of SP: Integration of Inductive Learning, Inference and Information Retrieval" (Ref. GR/G51565).

Correction

I would like to take this opportunity to correct a printing error in [25]. The formula which was given in Section 3.2 (p. 111) for the number of possible subsequences in a sequence of N symbols should, of course, have been $P = 2^N - 1$.

References

[1] Chaitin, G. J. (1987). *Algorithmic Information Theory*, Cambridge: Cambridge University Press.

[2] Chaitin, G. J. (1987). *Information, Randomness and Incompleteness: Papers on Algorithmic Information Theory*, Singapore: World Scientific Press.

[3] Chaitin, G. J. (1975). Randomness and mathematical proof. *Scientific American*, 232(5): 47-52.

[4] Cook, C. M. and Rosenfeld, A. (1976). Some experiments in grammatical inference. In J. C. Simon (Ed.), *Computer Oriented Learning Processes*, Leyden: Noordhoff, pp. 157-174.

[5] Harris, Z. S. (1954). Distributional structure. *Linguistics Today* 10: 146-62.

[6] Klop, J. W. (1992). Term rewriting systems. In S. Abramsky, D. M. Gabbay and T. S. E. Maibaum (Eds.), *Handbook of Logic in Computer Science, Vol. 2*, Oxford: Clarendon Press, pp. 1-116.

[7] Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission* 1(1): 1-7.

[8] Li, M. and Vitanyi, P. M. B. (1990). Kolmogorov complexity and its applications. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Amsterdam: Elsevier. Chapter 4, pp. 188-254.

[9] Minsky, M. (1956). Some universal elements for finite automata. *Automata Studies*, Princeton, NJ.: Princeton University Press, pp. 117-128.

[10] Pednault, E. P. D. (1991). Minimal length encoding and inductive inference. In G. Piatetsky-Shapiro and W. J. Frawley (Eds.), *Knowledge Discovery in Databases*, Cambridge Mass.: MIT Press.

[11] Post, E. (1943). Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics* 65: 197-268.

[12] Shannon, C. E. and Weaver, W. (1949). *The Mathematical Theory of Communication*, Urbana: University of Illinois Press.

[13] Solomonoff, R. J. (1964). A formal theory of inductive inference. Parts I and II. *Information and Control* 7: 1-22 and 224-254.

[14] Solomonoff, R. J. (1986). The application of algorithmic probability to problems in artificial intelligence. In L. N. Karnal and J. F. Lemmer (Eds.), *Uncertainty in Artificial Intelligence*, Elsevier Science, pp. 473-491.

[15] Stephen, G. A. and Mather, P. (1993). Sweeping away the problems that dog the industry? *AI Communications* 6(3/4): 213-218.

[16] Sudkamp, T. A. (1988). *Languages and Machines*, Reading, Mass.: Addison-Wesley.

[17] Uspensky, V. A. (1992). Kolmogorov and mathematical logic. *Journal of Symbolic Logic* 57(2): 385-412.

[18] Wallace, C. S. and Freeman, P. R. (1987). Estimation and inference by compact coding. *Journal of the Royal Statistical Society B* 49(3): 240-252.

[19] Wolff, J. G. (1977). The discovery of segments in natural language. *British Journal of Psychology* 67: 377-390.

[20] Wolff, J. G. (1982). Language acquisition, data compression and generalization. *Language & Communication* 2: 57-89. (Reprinted in Chapter 3 in [23]).

[21] Wolff, J. G. (1987). Cognitive development as optimisation. In L. Bolc (Ed.), *Computational Models of Learning*, Berlin: Springer-Verlag, pp. 161-205.

[22] Wolff, J. G. (1990). Simplicity and power - some unifying ideas in computing. *Computer Journal* 33(6): 518-534. (Reprinted in Chapter 4 in [23]).

[23] Wolff, J. G. (1991a). *Towards a Theory of Cognition and Computing*, Chichester: Ellis Horwood.

[24] Wolff J G (1991b). On the integration of learning, logical deduction and probabilistic inductive inference. *Proceedings of the First International Workshop on Inductive Logic Programming, Vienna do Costello, Portugal, 1991*, pp. 177-191

[25] Wolff, J. G. (1993). Computing, cognition and information compression. *AI Communications* 6(2): 107-127.

[26] Wolff, J. G. (1994). Towards a new concept of software. *Software Engineering Journal* 9(1): 27-38.

[27] Wolff, J. G. (1994). A scaleable technique for best-match retrieval of sequential information using metrics-guided search. *Journal of Information Studies* 20(1), 16-28.

[28] Wolff, J. G. (submitted for publication). Computing as compression: an overview of the SP theory and system.

[29] Wolff, J. G. (submitted for publication). Computing as compression: SP20.

[30] Wolff, J. G. and Chipperfield, A. J. (1989). SP: a computing system for the twenty first century. Technical Report CS-89-4.0, School of Electronic Engineering, University of Wales, Bangor.

[31] Wolff, J. G. and Chipperfield, A. J. (1990). Unifying computing: inductive learning and logic. In the Proceedings of Expert Systems 90: T. R. Addis and R. M. Muir (Eds.), *Research and Development in Expert Systems VII*, Cambridge: Cambridge University Press, pp. 263-276.

[32] Zvonkin, A. K. and Levin, L. A. (1970). The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms. *Russian Mathematical Surveys* 25: 83-124.