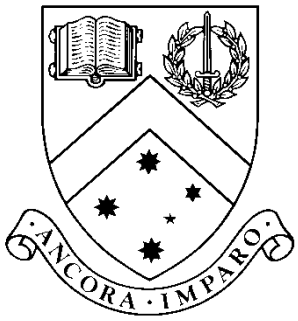
Approximating Algorithmic Complexity

by

Owen Rodda



Thesis

Submitted by Owen Rodda

in partial fulfillment of the Requirements for the Degree of

Bachelor of Software Engineering with Honours

in the School of Computer Science and Software Engineering at

Monash University

Monash University

November, 2003

Contents

Approximating Algorithmic Complexity

Owen Rodda, BSE(Hons)
Monash University, 2003

Supervisor: Assoc. Prof. David L. Dowe

Abstract

Algorithmic Information Theory is a field of study in which algorithm length is used as a complexity measure of the output produced. Determining the shortest algorithm for a given data set would effectively compress the data and measure its complexity. There is no constructive proof of the shortest algorithm for non-trivial examples, so in order to arrive at such an algorithm a different approach must be applied. This project demonstrates the viability of fitting algorithms to arbitrary data by generating computable functions as C source code from bit strings. The inverse conversion is also provided as a means for manual construction of algorithms in a framework suitable for comparison of algorithmic complexity. Using this approach a sufficiently accurate program could be found to match provided target data, measured by Minimum Message Length (MML) compression. The issue of considerably large search space is addressed by a distributed application architecture.

Approximating Algorithmic Complexity

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Owen Rodda
November 17, 2003

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Patterns in data can be described by algorithms. Identification of such algorithms is useful for compression as well as forming explanations for the data values. In situations where observed data has no obvious pattern, representative algorithms can be difficult to identify.

Algorithmic Information Theory and Minimum Message Length are two closely related theoretical fields of study which are of interest to computer scientists. Both relate to understanding the information content of data by description of the data in minimal representations.

The field of Algorithmic Information Theory incorporates a complexity measure called Kolmogorov Complexity, or Algorithmic Complexity, which is equivalent to the minimum length of an algorithm that outputs a particular bit string.

Minimum Message Length (MML) is a Bayesian technique which encapsulates data in a two-part message: a hypothesis or model of the data, and an encoding of the data based on that model. Models are ideally chosen such that the total length of the message is minimised; the more closely a model fits data, the better it expresses the patterns within and the shorter the resultant MML message.

MML can be considered a quantification of Occam's Razor such that it can be applied as an effective measure of program complexity (Patrick, 1996; Needham and Dowe, 2001). Programs performing the same task can therefore be objectively (given appropriate Bayesian prior probabilities) compared using MML to determine the more appropriate hypothesis.

MML models are usually restricted to a specific model class in order to minimise search space, limiting its usefulness in applications where the underlying pattern is not known and cannot be inferred. By generating and testing generalised computable function program code against data, the task of identifying useful algorithms can be automated.

Given enough attempts, a 'good' model could be found, determined by comparison of message length of the MML-encoded models and data. It is entirely possible that the resultant

code will not seem to have any logical sequence or explanation, implying that a high-level language representation is desirable for useful evaluation of results.

The largest obstacle to overcome is the search space of arbitrary algorithms. Non-trivial DNA computing problems, where candidate solutions are represented by DNA sequences, require a DNA pool whose physical equivalent would weigh more than the Earth (Hartmanis, 1995). To reduce this search space it will be necessary to limit resultant algorithms to a reasonable subset of those possible, and to apply as much computational power as possible.

This project aims to apply the principles of MML to provide, using distributed computing and automatic program generation, a method of finding appropriate algorithms for given arbitrary data sets. By developing distributed application software, accompanied by an informative World Wide Web site, the principles of Algorithmic Information Theory and Minimum Message Length encoding could be applied to a particular representation of C program code.

The following three chapters describe background to the project, including the purpose of Algorithmic Information Theory and MML and how they are related, followed by a discussion about distributed application architecture. Chapter 5 documents design decisions relevant to program representation. Chapter 6 discusses the resulting programs, and is followed by the conclusion with suggestions for possible future research.

Chapter 2

Algorithmic Information Theory

A string, defined as a sequence of bits, has the property of *complexity*, a measure of the randomness in that string. For example, the string 01010101010101010101 intuitively seems simpler than the string 00010101110101110101. An objective measure of complexity would help to decide the relative complexity of such strings, as well as indicate compressability — complex strings are random, and require more space when compressed than strings of lower complexity.

Algorithmic information Theory (AIT) provides such a measure, using algorithms to explain the complexity of a string. AIT is a field of study commonly attributed to similar yet independent discoveries by Solomonoff (1964), Kolmogorov (1965), and Chaitin (1966).

AIT is closely related to randomness and compression. Complex strings are random, and require more space when compressed than strings of lower complexity.

2.1 Solomonoff

Solomonoff's contribution to AIT was to develop a method of scientific inference and prediction based on selecting the smallest algorithm that explains the results of an experiment. As such Solomonoff's algorithms, binary encodings of theorems, when provided to a universal Turing machine (UTM) produce as a prefix of their output the observed results. A probability distribution of strings S is defined using summation over all finite input strings I which produce any string prefixed by S:

$$P_T(S) = \sum(2^{-|I|}). \tag{2.1}$$

Using this probability distribution, the algorithm with greatest posterior probability is selected as the explanation for observed behaviour (Solomonoff, 1964). Using the most

likely explanation, output from the selected algorithm can be extrapolated as a means to predict further behaviour.

2.2 Kolmogorov

The Kolmogorov complexity $K_T(S)$ of a string S is the length of the shortest binary string I which, provided as input to a universal Turing machine (UTM) T, produces exactly the string S (Kolmogorov, 1965). I is therefore an encoding of an algorithm that produces S. An example is the constant π; while its digits appear to be random, the existence of an algorithm to generate its nth digit (Bailey, Borwein and Plouffe, 1997) means that the algorithmic complexity of π is constant for any particular T. Some applications of Kolmogorov complexity are covered by Li and Vitányi (1997).

The choice of T is immaterial as two differing UTMs $T1$ and $T2$ can each be provided some constant-length program which causes it to act like the other. As the length of S increases, the constant interpreter length becomes less significant in calculation of the complexity of S (Gammerman and Vovk, 1999). That is, the difference between $K_{T1}(S)$ and $K_{T2}(S)$ has an upper bound of the larger of two constant interpreter lengths.

AIT can be generalised to programs in any Turing-complete programming language P instead of a Turing machine. Similarly to interpreters between different UTMs, interpreters can, by definition, be written to translate between Turing-complete languages and UTMs. In the context of high-level programming languages, source code length measured in bits is a measure of the complexity of the string produced by the corresponding program.

Finding an upper bound on the Kolmogorov complexity of a particular string is a simple matter of finding some program which generates the string as output. However, it is not possible to compute the algorithmic complexity of non-trivial strings; nor can it be known whether such a string is less complex than the shortest program yet identified which generates that string. These limitations are the focus of relevant work by Chaitin (1975).

2.3 Chaitin

Chaitin's contributions to AIT include practical examination of algorithmic analysis techniques (Chaitin, 1966) and proving that there is no way to determine the shortest program to generate a string and therefore there is no way to compute or identify the complexity of any non-trivial string (Chaitin, 1975). The proof is related to Gödel's incompleteness theorem, and states that within any formally defined numeric system the complexity of a string can only be proven if it is less complex than the system in which it is modelled. Chaitin modelled such a system in a LISP-like language of his own devising and found that all strings 410 characters more complex than the formal axiomatic system defining them are too long to be proven minimal (Chaitin, 1998).

Chaitin also states that most strings are random, given any reasonable definition of randomness. There is a probability of roughly one in 1000 that a string of length n has complexity less than $n - 10$. Thus, finding an algorithm to exactly produce some random string is not merely difficult, but often not possible.

2.4 Limitations of Algorithmic Information Theory

While AIT provides a definition of the complexity of a string, Chaitin's work shows that there is no way to determine whether a program is the shortest required for generating the string, and therefore the minimal complexity of a string cannot be determined.

The greatest problem with AIT is the search space involved in finding a minimal algorithm. The number of computable functions is immense, and since most strings are apparently random, finding an algorithm smaller than the target string is practically improbable. A 'best guess' complexity could be identified by manually creating an algorithm, but trial and error seems to be the only general approach.

Chapter 3

Minimum Message Length (MML)

3.1 Overview

Minimum Message Length (MML) was developed by Wallace and Boulton (1968), based on Shannon's theory of information. It is a technique used to encapsulate data in a two-part message: a hypothesis or model of the data, and an encoding of the data based on that model. Models are ideally chosen such that the total length of the message is minimised; the more closely a model fits data, the better it expresses the patterns within and, as a result, the shorter the resultant MML message. MML is therefore useful for inference (Wallace and Freeman, 1987). A similar approach called *Minimum Description Length* (MDL) has been developed by Rissanen (1978), and is compared in detail with MML in the Computer Journal special issue on Kolmogorov Complexity (Rissanen, 1999a; Wallace and Dowe, 1999a; Rissanen, 1999b; Wallace and Dowe, 1999b; Rissanen, 1999c; Wallace and Dowe, 1999c).

MML is a Bayesian approach, taking into account the prior probability of a model in determining the posterior. Bayes's formula

$$P(H|D) = \frac{P(H\&D)}{P(D)} \tag{3.1}$$

where H is the hypothesis (model), describes the likelihood of observed data D and can therefore be used in a negative log likelihood encoding based on the model.

Given appropriate priors, two models for an arbitrary string S can be objectively compared by encoding S as MML messages under both models and comparing the result.

The technique of determining the first part of the message, the hypothesis, is called MML inference (Wallace and Dowe, 1999a). If all hypotheses are assigned consistent priors then the optimal inference is that of greatest Bayesian posterior probability, and thus can be considered a formalisation of Ockham's razor, a heuristic attributed to William of Ockham

that can be paraphrased "if two theories explain the facts equally well then the simpler theory is to be preferred" but more accurately stated as "plurality shouldn't be posited without necessity" (Carroll, 2002). Empirical studies have supported the application of MML as an effective Ockham's razor (Needham and Dowe, 2001).

Since the search space for computable functions is vast (Hartmanis, 1995), MML inference is usually performed using a restricted model class to reduce that search space — for example, inference of coefficients of a polynomial to determine a line of best fit for given data.

MML is related to AIT as it provides a method of comparing the complexity of strings, either subjectively (where those strings are based on different languages or models) or objectively (where those strings are expressed in the same language) (Wallace and Dowe, 1999a).

3.2 Applications to Algorithmic Information Theory

It is not possible to prove that a string S generated by a program P is of a complexity greater than that of P (Chaitin, 1975). However, it is possible to find P with length less than $|S|$ provided such a program exists. This project will attempt to determine the viability of a 'brute force' approach to finding such programs by testing randomly generated programs for a supplied string S. The use of MML principles would differ from a pure algorithmic information theoretic approach by allowing those programs which do not exactly generate S to be accompanied by a second message part describing the difference between the algorithm and the target string.

3.3 Random Coding

A related concept is that of random coding (Dowe, 2003). Random coding requires both the sender and receiver of an MML message to use the same pseudo-random number generator θ. The sender selects a value i from the prior distribution over θ by sampling θ_1, θ_2, $\ldots \theta_i$. The message length can be costed based on the model θ_i: the first part of the message is simply an encoding of the natural number i (such as the log* encoding), and the second part of the message is simply

$$-\log f(data|\theta_i) \tag{3.2}$$

in order to determine which value of i minimises the message length. The trade-off is between selection of an appropriate value for θ_i, to minimise the data encoding, and the cost of stating i. This approach can be generalised to multi-dimensional distributions of θ.

Chapter 4

Distributed Applications

4.1 Overview

Many automated tasks require vast amounts of computing power. The Internet provides a method of connecting millions of computers worldwide such that they can work collaboratively on tasks which would take years to complete with today's technology, even a supercomputer. Several such projects have been deployed successfully to harness this computational power (Shirts and Pande, 2000; SETI@home, 1999–2003; Pande, 2000–2002; Pande, 2003; Distributed Computing Technologies, 1997–2003).

SETI@home, the distributed Search for Extra-Terrestrial Intelligence program, has demonstrated the viability of a collaborative large-scale computing project by accumulating over 1.5 million years' CPU processing time between the project's public launch in May 1999 and the end of July 2003.

An arguably more practical application of distributed computing is the *Folding@home* project, whose stated goal is "to understand protein folding, protein aggregation, and related diseases" (Pande, 2000–2002). The results of several protein folding studies have been collected and used as the basis of multiple scientific articles (Sorin, Rhee, Nakatani and Pande, 2003; Rhee and Pande, 2003).

4.2 Remote Procedure Calls

The most well-understood protocol on the largest computer network is the Hypertext Transfer Protocol (HTTP) on the World Wide Web (WWW), an application of the Internet. Many mechanisms have been created to communicate with HTTP servers, which are applications with implementation-independent resource location and delivery capabilities (Fielding, Gettys, Mogul, Nielsen, Masinter, Leach and Berners-Lee, 1999). There are three standard approaches to communicating encapsulated data with HTTP servers: XML-RPC, SOAP,

and REST (Bray, 2003). All are capable of transferring XML (eXtensible Markup Language) data, a highly structured human-readable data encapsulation standard (Bray, Paoli, Sperberg McQueen and Maler, 2000).

XML-RPC (XML Remote Procedure Call) is a simple, verbose protocol defining requests and response types for sending and requesting data structures comprised of nested anonymous primitives (Winer, 1999). It allows any HTTP-aware software to elicit a response analogous to function calls in a high-level programming language. Parameters are not explicitly named, instead requiring implementations to understand and interpret the data structures as received. Naming can be provided using a calling convention, however the additional overhead means that XML-RPC is best suited to simple applications.

SOAP (Simple Objects Access Protocol), another remote procedure call approach, is the result of work by the World Wide Web Consortium to address the need for RPC without the shortcomings of XML-RPC (Box, Ehnebuske, Kakivaya, Layman, Mendelsohn, Nielsen, Thatte and Winer, 2000). SOAP offers parameter naming and additional features such as additional data types. It is a correspondingly more complex application of XML, and is incompatible with XML-RPC.

Both XML-RPC and SOAP are most commonly used via *REST* (Representational State Transfer), a name retrospectively given to the operation of HTTP in a dissertation by Fielding (2000). The primary principle of REST is that the success of the WWW is due to a limited number of verbs — the HTTP request methods — applied to an infinite number of resources (URIs). RPC uses REST only for compatibility with HTTP, and extends the set of verbs available to cover required procedures. REST provides, at least, the flexibility of RPC at the expense of potentially greater programming effort, with the added benefit of taking advantage of HTTP conventions such as status codes.

Chapter 5

Program Representation

Given that the search space for algorithms or computable functions is potentially infinite, an effective search must aim to maximise the usefulness of those algorithms it checks. Source code, object code and machine code are all program representations with redundancy useful for efficiency, security or readability, among other reasons. When expressing algorithms, redundancy implies that multiple possible expressions of the same algorithm are possible, or that some combination of elements in that representation will result in a syntactically or semantically invalid program. By creating a representation with minimal redundancy, in which all sufficiently long combinations of data are interpreted as semantically valid programs, the generation of algorithms is simplified.

In the representation used, bit strings are self-delimiting such that only a prefix of long bit strings may be required. For any bit string comprising a complete program, addition of any suffix will not change the interpretation of the string.

5.1 High-level languages

High-level programming languages, such as C or Java, are widely understood and provide an abstraction of algorithms suitable for human interpretation. In contrast to low-level languages, such as an assembly language, high-level languages generally simplify the handling of architecture-specific code, representation of intermediate data, and logical structure of code. While both types of languages may be equally capable of representing an algorithm, the logic underlying high-level code is more easily interpreted by most human programmers than the logic underlying low-level code.

The redundancy of low-level program code is lower than that of languages with additional abstractions, and is designed to be highly similar to the machine code executed by the target architecture.

The representation devised was intended to encapsulate the high-level meaning of algorithms for later interpretation by human programmers. It has been designed in order to provide a mapping between C source and a minimally redundant serialisation of that source consisting of a string of binary digits, a bit string.

5.2 Computable functions

Much of the syntax available in C code is redundant, included primarily for convenience. For example, the `switch` statement can be simulated using multiple `if` statements, and `for` loops can be rewritten using `while` loops.

The available grammar was restricted to reduce this redundancy, limiting the algorithms to a specific subset of control structures and operations (see Table 5.1 on page 12). Using such a restricted grammar does not limit the expressiveness of C source code written by a human programmer provided that the conversion to a bit string can substitute equivalent generalised functionality for more specialised code.

5.2.1 Declaration semantics

The grammar itself does not sufficiently describe the content of a semantically-valid C source file, as there is no correlation between variable declarations and their usage. To ensure that only valid variables are referenced in the generated source code, a table of declared variable identifiers will be stored and subsequent variable references will refer to entries in that table.

To simplify variable handling, only the integer data type is allowed, with an optional single-dimensional array length. Higher dimensional arrays can be simulated with appropriate indices into a one-dimensional array. Integers were chosen as they are a general numeric format capable of representing Boolean truth values, characters, and signed integer numbers.

5.2.2 Identifiers

Variable names have no effect on program execution, merely providing a label to aid interpretation by programmers. The compressed representation omits variable names specified in source code, instead assigning sequential identifiers in generated code.

S	→	*declarations statement-list*
declarations	→	*declarations* “int” identifier *dimension default* ‘;’
declarations	→	ϵ
default	→	‘=’ integer
default	→	‘=’ ‘{’ integer ‘}’
default	→	ϵ
statement-list	→	*statement-list statement* ‘;’
statement-list	→	*statement-list if-statement*
statement-list	→	*statement-list while-loop*
statement-list	→	*statement-list block*
statement-list	→	ϵ
block	→	‘{’ *statement-list* ‘}’
if-statement	→	“if” ‘(’ *boolexp* ‘)’ *block else*
else	→	“else” *block*
else	→	ϵ
while-loop	→	“while” ‘(’ *boolexp* ‘)’ *block*
statement	→	*variable* ‘=’ *exp*
statement	→	“putchar” ‘(’ *variable* ‘)’
variable	→	identifier ‘[’ integer ‘]’
variable	→	identifier ‘[’ variable ‘]’
variable	→	identifier
dimension	→	‘[’ integer ‘]’
dimension	→	ϵ
boolexp	→	*variable* ‘<’ *exp*
boolexp	→	*variable* “<=” *exp*
boolexp	→	*variable* “==” *exp*
boolexp	→	*variable* “>=” *exp*
boolexp	→	*variable* ‘>’ *exp*
boolexp	→	*variable* “! =” *exp*
exp	→	*exp* ‘+’ *exp*
exp	→	*exp* ‘−’ *exp*
exp	→	*exp* ‘%’ *exp*
exp	→	*exp* ‘∗’ *exp*
exp	→	*exp* ‘/’ *exp*
exp	→	‘−’ *exp*
exp	→	*variable*
exp	→	integer
exp	→	‘(’ *exp* ‘)’

Table 5.1: Restricted C grammar

5.2.3 Integers

Integer values in this representation are stored in a format based on the efficient $\log_2^*$ (log star) encoding, which favours small integers and is capable of representing any positive integer. The algorithm used is modified to limit the range of values to those which can be stored in an integer. To generate the (base 2) representation of an arbitrary positive integer i:

1. Take the length, l, of the minimal bit-string representation of i
2. Prepend the minimal bit-string representation of $l - 1$ to i
3. Substitute 0 for the initial bit 1
4. Repeat steps 2 and 3 to encode the resultant value while l is greater than 1

Interpretation of random values works in reverse, sampling the number of binary digits to read until the leading bit of the sampled value is 1. To limit the representable numbers to the range of integers, once a length equal to or greater than the word size has been read the following word is used as the integer value, regardless of its initial bit value. Figure 5.1 illustrates how the binary number 1000101101110 would be represented.

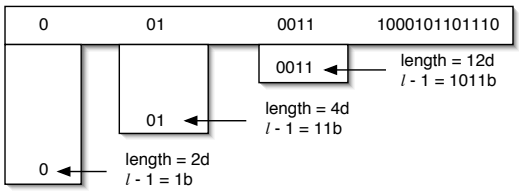


Figure 5.1: Log star encoding of 1000101101110

5.3 Initial values

If variables are evaluated before they have been assigned values, the outcome of their evaluation is undefined. A useful algorithm should operate consistently across multiple invocations, and therefore should not be dependent on the state of the machine's free memory when it is executed.

By initially assigning an arbitrary value to each variable in the program source code, values are consistent across executions. After conversion from bit string to C code, all variables and array elements will be assigned the initial value 0.

5.4 Memory access

Variable arrays were included in the grammar because they were considered important for non-trivial algorithms, but the use of array indices requires that appropriate memory bounds are adhered to. If bounds are exceeded, the program will either abort with a segmentation fault or proceed with undefined and potentially irreproducible results. Given the arbitrary selection of array indices in randomly generated source code, the bounds must be imposed to avoid frequent segmentation violations.

The representation chosen probabilistically selects either an integer within the required boundaries or a variable-based expression evaluated modulo the array length.

5.5 Output method

All output from the generated algorithms will be printed byte-wise to standard output. Byte-wise output provides a general output method capable of printing arbitrary strings, and preset data strings can be stored in arrays and output using a loop. The only function required by this representation is therefore "putchar".

5.6 Driver program

On its own the restricted grammar does not describe a complete C program. Conversion from bit string to C code must therefore wrap the result in a driver function, called upon program execution, as well as any required header declarations. Since the grammar allows for one standard function and one function only, the `stdio.h` header is the only necessary header.

5.7 Execution timeout

Since the halting problem is insoluble, generated programs may or may not halt, and given enough attempts such a program will eventually be encountered.

To avoid programs with infinite processing time, a strict time limit is imposed on the running time of each compiled program. The time limit may prematurely terminate a program that would otherwise complete with the required result, and therefore the limit must be selected based on a realistic allowance. An appropriate time limit also has the added benefit of avoiding impractically slow solutions.

5.8 Prior probability

As MML is a Bayesian technique, its application relies on selecting a prior probability for the first part of the message, the model. In the representation used for this project each production rule in the grammar has a constant, context-independent probability[1] with the exception that statement and declaration lists cannot be empty. These probabilities were chosen arbitrarily to encourage reasonable relative proportions of each code construct, but do not fully take advantage of possible context-dependent optimisations which could include:

- Usage of every declared variable
- Probabilities dependent on depth of control structures
- Favouring update of variables used in loop conditions

These optimisations would reduce the compressed string length and increase the likelihood of a useful program, however they are not necessary given that they do not improve upon the output grammar expressiveness. A universal prior allowing the expression of all computable functions is sufficient, just as the choice of Turing machine is unimportant when dealing with Kolmogorov complexity using Turing machines.

[1] Logical groupings such as parentheses and blocks are generated where appropriate and are not assigned a probability in situations where they are required.

Chapter 6

Results and Discussion

6.1 Client-side applications

The client-side suite of applications that were produced consisted of two conversion programs, to map between bit strings and C code; a Perl script to automate the generation, execution and comparison of output; and a trivial program to produce bit strings of a specified length, drawn from the random source `/dev/random`. Figure 6.1 shows an overview of interaction between the conversion programs.

Client files are stored in multiple directories for convenience:

- "./": Client Perl script and configuration
- "./src/": C, C++, Bison and Flex source files
- "./pending/": Bit strings to be examined
- "./deferred/": Bit strings whose compiled code did not terminate
- "./target/": Strings with which to compare output
- "./outputstrings/": Output from successful program runs

All programs were developed to run on Unix-based operating systems, using the GNU utilities gcc, g++, make and bison, also requiring perl and flex. Several perl libraries – LWP::UserAgent, XML::Simple, Proc::Background and their dependencies – are required, but can be made available for download and installation from the Web site.

6.1.1 Bit string to C source conversion

The program `bs2c` was implemented using object-oriented C++ to store the parse tree. It draws input as raw characters from standard input until a complete program has been

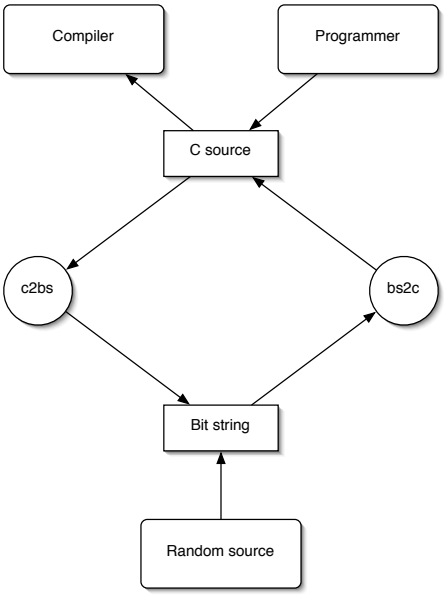


Figure 6.1: Conversion overview

created. If an input string is not of sufficient length to describe a complete program, `bs2c` will exit with the error "Input string too short."

The C++ classes were shared between both conversion applications, and represent distinct nodes in both source code and bit string representations. Each object in the parse tree is capable of reading its parameters in bit string representation, and producing output in both formats.

To construct the parse tree, bits are sampled in groups from the input stream as required and examined as integers to probabilistically determine the following child or sibling nodes. The entire output is wrapped in a simple driver function which encapsulates the generated code – for an example, see Listing 6.1.

6.1.2 C source to bit string conversion

The program `c2bs` was implemented as a C tokenising scanner generated with Flex, and a C++ parser generated by Bison. C source code matching the restricted grammar is read from standard input and used to construct a parse tree. Conversion requires that the code is not wrapped in the driver function, meaning that output from `bs2c` cannot be directly processed by `c2bs`.

Variables are collected into a single declaration block at the beginning of the driver function, and initialised to 0 regardless of their initialisation values in the input file.

Table 6.1: C++ node classes

ProgramNode (base class)
AssignmentNode
BlockNode
ConditionalNode
FunctionNode
IntegerNode
LoopNode
OperationNode
VariableNode
VariableTable

After construction of the parse tree, the bit string is written to standard output, as a sequence of bytes, with any incomplete bytes padded with zeroes. Multiple conversion between the two formats will increase the size of a bit string only when variable arrays are referenced, since C source output adds a modulo operation to limit array indices.

Conversion from C source to bit strings allows programmers to provide their own attempts at identifying patterns in data simply by adding the bit string representation of their code to the 'pending' directory for processing.

6.1.3 Perl client driver

A Perl script, `client`, is responsible for the automation of client-side activities. Given a system with the appropriate libraries available, the client script alone is enough to download, compile, and run all of the other applications used on the client-side. The Perl client algorithm is listed in detail in Figure 6.2.

Comparison of the output string with target strings was performed by examination of the binary exclusive OR (XOR) of the two bit strings. Where the output from generated programs was shorter than the target string it was repeated until the required length was achieved; programs producing no output were discarded. Output exceeding the target string length was truncated. The resultant string was then compressed using the command-line tool `gzip` and its compressed length used as the length of the second part of the MML message. An exact match would be highly compressible, and the compressed length gives a measure of the 'distance' from one string to the other.

The first part of the MML message can be costed by running the relevant code through `c2bs` and examining the output length. Combined with the length of the compressed second part, the total length represents the evaluated complexity of a program which describes exactly the target string, and can be sent to the server for evaluation. Automation of this part

Listing 6.1: Typical generated C source output (formatted for readability)

```
#include <stdio.h>

void runcode() {
        int v1 = 0;
        int v2 = 0;
        int v3 = 0;
        int v4[9112] = {0};
        int v5 = 0;
        while (v2 >= v3) {
                putchar(v4[748 % 9112]);
                if (v1 < v4[1190 % 9112] * v3 - 10) {
                        putchar(v5 * v5);
                        putchar(1);
                        v5 = v1;
                        v3 = v3;
                        v2 = v3;
                }
                if (v1 < 4) {
                        putchar(v2);
                }
        }
        putchar(v2);
        v5 = 1;
        v3 = 3;
        v4[v2 * 28 % 9112] = 3 / v3;
        v2 = 2;
        putchar(247);
        v3 = 2;
}

int main(int argc, char *argv[]) {
        runcode();
        return 0;
}
```

of the process was not completed due to unfavourable results from generated programs, described below.

Execution time limit was implemented in the client as a modifiable configuration option with a default timeout of ten seconds. Output processing was not initiated until after the program had terminated; in that time, hundreds of megabytes of disk space could be consumed by infinite output loops.

6.2 Server-side scripts

All server-side functionality was implemented as a collection of PHP scripts producing XML output. The "home page" document contained at the base URI was an XHTML document containing information for users and a hyperlink to the client application so that new users can obtain the software. A link, such as that shown in Listing 6.2 and invisible to standard Web browsers, within the source code directed the client application to a separate resource, specifically for automation and resource discovery, with links to the latest client, libraries, target bit strings and a reporting location. Using the link, clients could be directed to a new home page should the project move.

Listing 6.2: Automation link

```
<link rev="automation" type="text/xml" href="discovery.xml" />
```

Listing 6.3: Example discovery resource

```
<resources>
        <target>http://localhost/string/</target>
        <result>http://localhost/string/</result>
        <client version="0.18">http://localhost/client.pl</client>
        <libraries>http://localhost/lib.tar.gz</libraries>
        <source version="0.18">http://localhost/src.tar.gz</source>
</resources>
```

The *result* resource handles dissemination and collection of bit strings, responding to requests based on the request type:

- GET requests to retrieve the current target string or strings
- POST requests to submit bit string representations of programs
- PUT requests to add new target strings
- DELETE requests to remove previous target strings

1. Check for presence of modules
    - Get required modules if possible
2. Locate home page
3. Check client version (not implemented)
    - Download new client
    - Swap scripts
    - Execute new client
4. Check for programs
    - Download source
    - *Make* if necessary
5. Look for existing/pending strings
6. Check tasks
    - Upload previous attempts
    - Download new target string(s)
7. Program generation cycle
    (a) Generate random string
    (b) Convert string to C code
        - Reformat if *indent* available

    (c) Compile C code
    (d) Execute compiled program
        - If timeout exceeded, store string for later analysis
        - If no output, discard string

    (e) Compare output
        - Report improvements

Figure 6.2: Client script algorithm

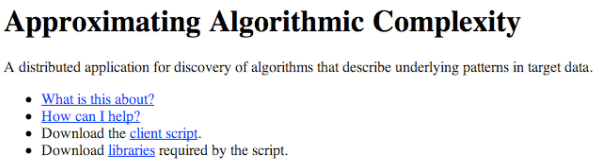


Figure 6.3: Simple project World Wide Web home page

Of the four, only GET and POST were implemented during this project. The request and response content for both used a simple XML representation; Listing 6.4 demonstrates the format of the result from a GET request on the resource.

Listing 6.4: Example target string resource

```
<strings>
        <target name="pi"
                hash="77314c6765d09658e4c9b6d60d43acd5"
                status="current"><![CDATA[31415926535...]]></target>
        <target name="unknown-1"
                hash="49b2e0027abb21339e3d12947a4bb0e2"
                status="current">&#018;&#023;...</target>
</strings>
```

6.3 Observations

The large majority of programs generated using `bs2c` are either particularly long, in terms of both the bit string and `C` source code, or contain expressions with a disproportionately large amount of operators.

Output from most generated programs ranged in length from zero bytes to almost three hundred megabytes. Large file sizes are the result of infinite loops containing output statements.

A sample run of one thousand programs, using client defaults, resulted in 776 failing due to insufficient input length, 30 producing no output, and 194 producing at least one character. Of those which output data, twelve failed to terminate, and the remaining 182 output between one and ten bytes as shown in Figure 6.4.

Short output from these generated programs suggests that significant optimisations must be made in order to create a useful system. Identifying patterns in a short target data set is a trivial problem better suited to other forms of analysis, yet given these results it appears to be unlikely that a high proportion of terminating programs will provide output of a length suitable for non-trivial target strings.

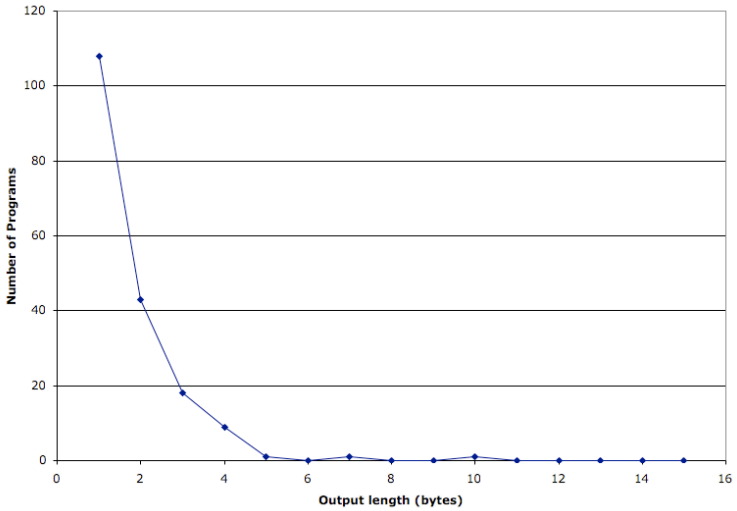


Figure 6.4: Output length of sample generated programs

Due to these poor results from generated programs, and to time constraints, no algorithms were tested against any specific target data.

Chapter 7

Conclusion

A bit string representation of C code using a restricted grammar was devised, and programs were created to successfully convert between C source and bit strings. The conversion was based on a simple probabilistic model with little reliance on context, and provided a language sufficiently expressive to represent computable functions.

To address the issue of search space, a distributed framework was created allowing for generation, execution and evaluation of algorithms on multiple clients simultaneously, where communication to the server was required only after evaluation of a successful algorithm. The client environment provided for manual construction of programs in addition to automation, and the HTTP-based server provided a point of access to information to both the automated client and to human users.

Application of the client programs indicated that significant changes to the conversion programs must be made in order to achieve a valuable and effective system. Unfeasibility of algorithms generated using the prior probability of the selected model indicates that optimisations in the conversion routines would greatly benefit the system, increasing the likelihood of producing a reasonable approximate algorithm.

7.1 Future research

As mentioned above, the conversion process wold benefit from a context-dependent constructive conversion from bit strings to C source, and a corresponding change in the inverse conversion. Observations indicate that the relative prevalence of control structures such as 'while' loops in generated source code was variously and disproportionately under- and over-represented, due to the nesting behaviour of such constructs. It is possible that a change in probabilities assigned to each production in the grammar would sufficiently change the output; however, a more effective method would be to empirically examine a wide range of C source code files to determine a more realistic balance between nodes. Inclusion of spatial

information such as a bias on printing towards the end of a program, and proper utilisation of variables throughout expressions, would both increase compression and improve the usefulness and variety of output produced. A particular focus should be on the use of loop invariants to improve the efficacy of generated loop expressions and associated statements.

Further improvements could potentially be made by adding to the grammar. While simplicity helps to reduce redundancy, addition of features such as functions to the grammar could increase the likelihood of emergence of a powerful algorithm due to the ability to create recursive programs.

Finally, an alternative bit string sampling method could be implemented. Instead of sampling a constant number of bits from the random source, a variable bit length for alternative outcomes would ensure that those productions with high prior probability take up a minimum amount of space, maximising the compression benefit.

References

Bailey, D., Borwein, P. and Plouffe, S. (1997). On the Rapid Computation of Various Polylogarithmic Constants, *Mathematics of Computation* **66**(218): 903–913.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S. and Winer, D. (2000). Simple Object Access Protocol (SOAP) 1.1, Note.
URL: *http://www.w3.org/TR/2000/NOTE-SOAP-20000508*

Bray, T. (2003). The SOAP/XML-RPC/REST Saga, Chap. 51, World Wide Web site.
URL: *http://tbray.org/ongoing/When/200x/2003/05/12/SoapAgain*

Bray, T., Paoli, J., Sperberg McQueen, C. M. and Maler, E. (2000). *eXtensible Markup Language (XML) 1.0*, second edn, World Wide Web Consortium.
URL: *http://www.w3.org/TR/2000/REC-xml-20001006*

Carroll, R. T. (2002). William of Ockham (1285-1349), In The Skeptic's Dictionary, World Wide Web site. (last accessed July 29, 2003).
URL: *http://www.skepdic.com/occam.html*

Chaitin, G. J. (1966). On the Lengths of Programs for Computing Binary Sequences, *J. Assoc. Comput. Mach.* **13**: 547–569.

Chaitin, G. J. (1975). Randomness and Mathematical Proof, *Scientific American* **232**(5): 47–52.
URL: *http://www.cs.auckland.ac.nz/CDMTCS/chaitin/sciamer.html*

Chaitin, G. J. (1998). Elegant Lisp Programs, *in* C. S. Calude (ed.), *People and Ideas in Theoretical Computer Science*, Springer-Verlag Singapore, pp. 32–52.
URL: *http://www.cs.auckland.ac.nz/CDMTCS/docs/calude.html*

Distributed Computing Technologies (1997–2003). distributed.net, World Wide Web site.
URL: *http://distributed.net/*

Dowe, D. L. (2003). C. Wallace's ideas on random coding, Informal discussion.

Fielding, R., Gettys, J., Mogul, J., Nielsen, H., Masinter, L., Leach, P. and Berners-Lee, T. (1999). *Hypertext Transfer Protocol – HTTP/1.1 (RFC 2616)*, Internet Engineering Task Force.
URL: *http://www.w3.org/Protocols/rfc2616/rfc2616.html*

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*, PhD thesis, University of California, Irvine.
URL: *http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm*

Gammerman, A. and Vovk, V. (1999). Kolmogorov Complexity: Sources, Theory and Applications, *The Computer Journal* **42**(4): 252–255.
URL: *citeseer.nj.nec.com/342525.html*

Hartmanis, J. (1995). On the Weight of Computations, *Bulletin of the European Association for Theoretical Computer Science* **55**: 136–138. (referenced material based on extract).

Kolmogorov, A. N. (1965). Three Approaches to the Quantitative Definition of Information, *Prob. Inform. Transmission* **1**: 1–7.

Li, M. and Vitányi, P. M. B. (1997). *An Introduction to Kolmogorov Complexity and Its Applications*, 2nd edn, Springer, New York.
URL: *citeseer.nj.nec.com/li97introduction.html*

Needham, S. L. and Dowe, D. L. (2001). Message Length as an Effective Ockham's Razor in Decision Tree Induction, *8th International Workshop on Artificial Intelligence and Statistics*, pp. 253–260.

Pande, V. (2000–2002). Folding@home, World Wide Web site. (last accessed November 1, 2003).
URL: *http://folding.stanford.edu/*

Pande, V. (2003). Genome@home, World Wide Web site. (last accessed November 1, 2003).
URL: *http://genomeathome.stanford.edu/*

Patrick, J. (1996). A Minimum Message Length (MML) Model for Software Measures, *Proc. Information, Statistics and Induction in Science Conf. (ISIS'96)*, World Scientific, Melbourne, Australia, pp. 385–394.

Rhee, Y. M. and Pande, V. S. (2003). Multiplexed-Replica Exchange Molecular Dynamics Method for Protein Folding Simulation, *Biophysical Journal* .

Rissanen, J. (1978). Modelling by shortest data description, *Automatica* **14**: 465–471.

Rissanen, J. (1999a). Discussion of paper 'Minimum Message Length and Kolmogorov Complexity' by C. S. Wallace and D. L. Dowe, *The Computer Journal* **42**(4): 270–283.

Rissanen, J. (1999b). Hypothesis Selection and Testing by the MDL Principle, *The Computer Journal* **42**(4): 260–269.

Rissanen, J. (1999c). Rejoinder, *The Computer Journal* **42**(4): 343–344.

SETI@home (1999–2003). The Search for Extra-Terrestrial Intelligence @Home, World Wide Web site. (last accessed November 1, 2003).
URL: *http://setiathome.ssl.berkeley.edu/*

Shirts, M. and Pande, V. S. (2000). Screen Savers of the World Unite!, *Science* **290**(5498): 1903–1904.
URL: *http://folding.stanford.edu/papers/SPScience2000.pdf*

Solomonoff, R. (1964). A Formal Theory of Inductive Inference, Part I, *Information and Control, Part I* **7**(2): 1–22.
URL: *http://world.std.com/ rjs/1964pt1.pdf*

Sorin, E. J., Rhee, Y. M., Nakatani, B. J. and Pande, V. S. (2003). Insights Into Nucleic Acid Conformational Dynamics from Massively Parallel Stochastic Simulations, *Biophysical Journal* .

Wallace, C. S. and Boulton, D. M. (1968). An Information Measure for Classification, *Computer Journal* **11**(2): 185–194.

Wallace, C. S. and Dowe, D. L. (1999a). Minimum Message Length and Kolmogorov Complexity, *The Computer Journal* **42**(4): 270–283.
URL: *citeseer.nj.nec.com/wallace99minimum.html*

Wallace, C. S. and Dowe, D. L. (1999b). Refinements of MDL and MML Coding, *The Computer Journal* **42**(4): 330–337.

Wallace, C. S. and Dowe, D. L. (1999c). Rejoinder, *The Computer Journal* **42**(4): 345–347.

Wallace, C. S. and Freeman, P. R. (1987). Estimation and Inference by Compact Coding, *J. R. Statist. Soc B* **49**(3): 240–265.

Winer, D. (1999). XML-RPC Specification, World Wide Web site. (last accessed October 22, 2003).
URL: *http://www.xmlrpc.com/spec*

Appendix: Client and Server Setup

Required files may be attached to this document or archived at a location available from `http://www.csse.monash.edu.au/hons/`.

Server

Running the provided server code requires an Apache Web server running PHP with the MultiViews enabled. To install and use:

1. Create a new directory on the Web server (or use the root directory).
2. Extract the contents of `server.tar.gz`.
3. Move the contents of the extracted directory to the created Web server directory.
4. Modify `discovery.xml.php` to include an appropriate base address.

Client

Client code was developed and tested on Macintosh PowerPC architecture, but should run on other architectures.

To use a client with access to the Web server:

1. Browse to the home page
2. Download the `client.pl` file into a new directory
3. Change permissions to allow execution of the `client.pl`
4. Run the `client.pl` file from the new directory
5. Provide the home page URL when prompted

You may be required to install additional Perl modules; these can be placed within the directory `./lib/` under the location where the client script is installed.

To execute the conversion programs without access to the Web server:

1. Extract `client.tar.gz` to a new directory
2. From the new directory, run the GNU `make` utility

Without a Web server the conversion and execution is not automated, but the applications `bs2c` and `c2bs` can be manually invoked.