How to publish algorithms reproducibly and a new elegant algorithm for sequential experiments

HAROLD THIMBLEBY, DAVID WILLIAMS, University of Swansea, Wales

This paper presents a very short, elegant and portable algorithm for finding Euler cycles that has important applications in the design of sequential experiments to efficiently control bias, drift, random error, carry-over and other effects.

The algorithm is presented in C and is written in a clear style to simplify porting to other languages.

To explain and present the algorithm rigorously, we use and describe an original and powerful "reverse literate programming" technique that generates the executable code directly from this manuscript for publication. The technique ensures that both the executable code and the published paper are synchronised, preventing transcription errors and making publishing correct code easier. The approach increases transparency, reproducibility, and accessibility — and, more generally, encourages authors to publish details of working embodiments of algorithms rather than abstract or simplified descriptions.

The approach can be used for any formal material: mathematics, proofs, or — as in the present case — algorithms and programs. It can be used in papers (as here), in reports and books and even, with the same advantages, in student work.

Keywords: Euler cycle algorithm; de Bruijn sequence; Combinatorics; Experimental design; Literate programming; Reproducibility.

CCS Concepts:•**Applied computing** → **Annotation;**•**Theory of computation** → *Graph algorithms analysis;*

ACM Reference Format:
[Submitted] H. Thimbleby, and D. Williams, How to publish algorithms reproducibly and a new elegant algorithm for sequential experiments *ACM Trans. Soft. Eng. Methodology* 1, 1, Article 1 (May 2016), 24 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Getting software to work is often harder than it ought to be. When we needed a standard "off the shelf" algorithm we found it was quicker to invent a new one from scratch than get published algorithms to work.

This paper first introduces our brief, elegant new algorithm and its use and applications in experimental methods. It was hard to find a working solution in the literature — even when we had searched both peer reviewed publications and textbooks — and we suspect this is a widespread problem. Therefore we propose a new approach to help write more reproducible publications about algorithms, for which we built a tool **relit** (so called because it implements "reverse literate programming"). In fact, we used **relit** in the present paper: the new algorithm we present in this paper works exactly as shown.

This research was funded by EPSRC under grant no. [EP/L019272/1].
Authors' address: H. Thimbleby and D. Williams, University of Swansea, SWANSEA, Wales, SA2 8PP, UK.

DOI: 0000001.0000001

Finally, we relate our new, lightweight tool-based approach to Richard Feynman's exhortation to avoid what he calls "cargo cult science": when we have the right tools to help us, his challenge becomes easy. We show how our approach makes publishing working software easier and more likely. It helps improve the quality of not just of publications but also of the underlying algorithms themselves: authors can now easily improve their algorithms and their papers as a tightly integrated process.

2. ALGORITHMS FOR SEQUENTIAL EXPERIMENTS

In a sequential experiment, such as generation of sensor calibration curves, a number of random and systematic errors may occur. Errors can include: bias, drift of sensor readings with time, and carry-over effect where the previous reading influences the next reading. Good experimental design therefore uses a sequence of calibration values arranged in such a way the sequence will normalise and cancel out random and systematic errors. Ideally the sequence will be as short as possible in order to minimise unnecessary work and expense in the collection of data.

The problem may be illustrated by a familiar example. We might want to know which of N types of wine tastes best, but, as is well known, the flavour of a wine is affected by the last wine just tasted. We therefore want to design a systematic experiment for wine tasting that tries every sequential combination of pairs of the set of N wine types available in our cellar, and of course we want the shortest such cycle, because experiments with wine are expensive.

For the sake of concreteness, suppose we have $N = 3$ types of wine in our cellar, specifically, say, Merlot, Pinotage and Shiraz. If we have just drunk Pinotage, then there are three possible experiments to do next: to drink Merlot next, Shiraz next, or of course to drink Pinotage again. If we drink Pinotage, then the next experiment should probably be to assess Merlot or Shiraz, since we already know what Pinotage after Pinotage tastes like.

With only three wines, the best sequence of experiments is not too hard to work out, but in general with lots of wine it becomes much harder (especially if we start the wine tasting before we have finished working out the right sequence).

For simple cases the problem can be solved by hand by drawing a graph with one vertex for each of the N types of experiment (e.g., testing a type of wine) with N^2 arrows to represent each possible sequence of two experiments. This creates a *complete graph*, usually denoted K_N. Figure 1 (on page 6) shows the complete graph for $N = 3$. (The background in graph theory is explained in section 2.1 below.)

A sequence of several experiments forms a path following arrows in this graph. In our problem we want to find a shortest path that follows every arrow at least once. It is a standard graph theory result for a complete graph that a shortest sequence only needs to follow each arrow exactly once, and it will also end up where it started — obviously, if it followed any arrow more than once, it would be repeating an experiment and take longer. An optimal sequence is called an *Euler cycle*, after Leonard Euler who first studied paths in graphs (but not sequential experiments). Our problem now reduces to finding an Euler cycle algorithm.

Our problem, then, is to write a computer program to generate an efficient sequence of experiments for the general case. Fortunately, finding Euler cycles is a well-known programming problem, which has been solved many times. Unfortunately, it turns out that programming textbooks typically leave this problem as an exercise for the reader, and that does not help if your aim is to do a sequential experiment, rather than learn how to program algorithms!

The definitive algorithms textbook by Cormen, Leiserson, Rivest and Stein just says:

> **Exercise 22-3** [. . .] Describe an $O(E)$-time algorithm to find an Euler tour of G if one exists. (*Hint:* Merge edge-disjoint cycles.) [Cormen et al. 2009]

Elsewhere Cormen estimates that writing up the solutions to the book's exercises would run to between 2,000 and 3,000 pages [Cormen 2016]: it is just not going to happen. Other books do not show answers to exercises in case students might cheat. At least it is pretty clear where we stand.

But worse than not providing an algorithm, in many textbooks and published papers, the Euler cycle algorithm — if presented at all — is presented as a described in high-level English, as a sketch, or in a simplifying pseudo-language. It is also easy to find numerous lecture notes online that describe the Euler cycle algorithm in English.

While English may be sufficient to explain some principles or to estimate the time complexity of the algorithm or other properties, and to do other things programmers are interested in, it is inadequate to convert into a working program in a real, executable, language such as C or Java. To get a working program, a lot of detail needs careful consideration: how should a graph be represented in a language with type checking, for instance? If vertices are numbered, are they numbered from 0 or 1? Or should you use a standard package, and then have to convert the pseudo-program into the programming conventions of the package? Unfortunately, the necessary detail makes explanations unwieldy. In the worst case, the detail never existed, and the presented algorithm is not even an abstraction of anything that was ever executable.

Furthermore, Euler cycle programs assume the graph is arbitrary, and there are different algorithms for directed, undirected and mixed graphs. In our case, the graph happens to be both directed and complete, and we discovered that these facts can be used to simplify the problem. In fact, inventing and implementing a new algorithm for our special case took less time than correctly implementing a standard algorithm.

Our complete working program is 19 lines of code; moreover, the core algorithm is only 9 lines. (The code is given in section 3.3.) This compares favourably to a quality reference algorithm [Sedgewick et al. 2015] that is 22 lines of code (plus 49 lines of test code), though it will not work "out of the box" without also finding and compiling it with the various files it depends on — the total code needed runs to 493 lines, about 25 times longer.[1]

This reference code also requires further work to edit the Java and local Java environment to compile it, as well as code to define a complete graph and print the desired solution (details that are already included in our algorithm). Furthermore, while Java may be a fine language, it is hard to translate algorithms written in it into other languages; whereas our code, written in basic C, is easy to translate into any language that has arrays (Fortran, PHP, JavaScript, Mathematica, Matlab, Java itself . . .), which will be a considerable advantage for experimenters who are not familiar with Java

The Java reference algorithm is sophisticated and no doubt ideal for teaching purposes, but there is an evident separation of the published book text [Sedgewick and Wayne 2011] from the published program [Sedgewick et al. 2015], a separation that allowed the code to develop into a sophisticated program independently of the published book. Unsurprisingly, the book omits the code altogether: it has to be downloaded from the web instead. Of course, this is entirely defensible: the web code has many details that go far beyond what is needed in a book — the code includes unit tests and examples, which would be tedious and distracting to explain in a book. Even when code from a working program is published, unknown details in the support code that is not published may be critical to getting it to work properly.

[1] Line counts ignore comments and blank lines. No attempt has been made to "squash" code to save lines.

It is interesting to note the polarising force of positive feedback when writing about algorithms:

— When code cannot be seen by the reader, the author (as a good programmer) is under pressure to add features so it can do anything that a reader might be anticipated to want. There is thus a natural tendency to add features, which tends to make the code more complex, which further justifies keeping it out of sight ...
— When code is brief and visible, the reader can adapt what they can see themselves: the author has no pressure to add features. When code is visible, there is a natural tendency to improve and clarify it, which further justifies keeping it visible ...
— The third possibility is that the author manages the complexity by abstracting the algorithm: this makes the paper concise, but conceals details the author finds irrelevant to their immediate goals.[2] As "irrelevance" is abstracted away, the author thinks less and less about the reader's possible wider needs, which further focuses the paper on the chosen abstraction ...

It is clear that normal writing and publishing practices combined with the intricacies of managing working programs do not align well with the goal of finding reproducibe algorithms. The goals and priorities of publishing are not the same as the goals of science; the pressures of publishing, whether papers or books, conspire to compromise reproducibility.

2.1. Graph theory background and applications

Leonard Euler effectively invented graph theory in 1736 with his solution to the famous Königsberg[3] bridge problem [Euler 1736; Hopkins and Wilson 2004]: is it possible to walk a cycle across each of the seven bridges in Königsberg exactly once? In modern graph terminology, a bridge is an *edge* and the land a bridge ends on is a *vertex*; if bridges are one-way to traffic, then the graph is a *directed graph* as opposed to an *undirected graph*. An *Euler* (or *Eulerian*) *cycle* is a walk that traverses each edge of a graph exactly once, starting and ending at the same vertex.

This paper is concerned specifically with directed graphs. A *complete* directed graph is a graph in which each pair of distinct vertices is connected by two directed edges, one in each direction. A complete graph of 3 vertices is shown in figure 1.

> *Notation*. We use $u \to v$ for the directed edge (arrow) connecting vertex u to vertex v, and $u \rightsquigarrow v$ for a directed path, consisting of one or more edges, connecting u to v.

A *cycle* is a path that starts and ends at the same vertex.

A directed graph is *(strongly) connected* if it contains a directed path $u \rightsquigarrow v$ and a directed path $v \rightsquigarrow u$ for every pair of vertices u, v. Since edges are paths of length 1, complete graphs are strongly connected.

A *bridge* (in graph theory, as opposed to a bridge over a river) is an edge that if it was deleted, the graph would no longer be connected.

A (k, n) *de Bruijn sequence* is a cyclical list of length k^n which contains k unique symbols, arranged so that every permutation of overlapping sublists of length n occurs exactly once. For example, 0011 is a $(2, 2)$ de Bruijn sequence using 0 and 1 as the symbols, since all the length 2 sublists 00, 01, 11, and 10 (wrapping around) each occur exactly once, and in this order. Although the sequences were named after Dutch mathematician Nicolaas Govert de Bruijn [de Bruijn 1946], they had been previously described in the 19th century [Fleury 1883; Sainte-Marie 1894].

[2] Few papers talk about more than one of: complexity, correctness, implementation, security, usability ...
[3] Königsberg was in Prussia and is now Kaliningrad in Russia.

The Sanskrit poet Pingala employed a $(2, 3)$ de Bruijn sequence over $1,000$ years ago to permute poetic meters and drum rhythms [Hall 2015; Knuth 1998a]. Modern applications of de Bruijn sequences include: gene-sequencing, magic tricks, optimal strategies for opening combination locks, and cryptography (including generation of one-time pads, and the Data Encryption Standard algorithm). Since each sublist occurs exactly once in the sequence, two-dimensional de Bruijn arrays may be used to identify the position of industrial robots on a warehouse floor, or the position of an infrared-sensing pen on specially marked paper [Diaconis and Graham 2012].

Fisher proposed the application of randomised Latin Squares to balance and equalise sampling error and bias in the design of experiments into soil fertility [Fisher 1925]. In a similar way, if a sequence of experimental tests is prescribed by a randomised $(N, 2)$ de Bruijn sequence, every element in the series will be tested N times in random order, and will be immediately preceded by every other element in the series in the set exactly once. This approach allows the most efficient means of testing multiple items balancing and equalising the effects of systematic bias, drift, serial carry-over effects, hysteresis and random error. Thus a simple algorithm to generate randomised de Bruijn sequences has applications in experimental design in many areas of scientific and statistical research, and can also be used in signal processing to improve the signal to noise ratio, for instance in Magnetic Resonance Imaging (MRI) scan [Aguirre et al. 2011].

An Eulerian cycle may be used to generate de Bruijn sequences [Fleury 1883; Hierholzer 1873]. An Eulerian cycle of a complete graph with N vertices follows in order every edge $u \to v \to \ldots$ exactly once for all u and v, which is precisely the definition of an $(N, 2)$ de Bruijn sequence. More generally, a *de Bruijn graph* is a graph whose Euler cycle generates the corresponding de Bruijn sequence. As a special case, the complete graph is a de Bruijn graph; Good [Good 1946] gives further examples. Other approaches for generating de Bruijn sequences include feedback shift registers and genetic algorithms [Turan 2011; Knuth 1998a].

2.2. Classic Euler cycle algorithms

Many deceptively simple algorithms to generate Eulerian cycles have been described. These two classic algorithms were invented well before modern computers:

Hierholzer's algorithm. [Hierholzer 1873] first finds all cycles in the graph then merges them together.
Fluery's algorithm. [Fleury 1883] follows a path successively choosing edges to delete at each vertex by first choosing any edge that is not a bridge, and finally choosing the bridge when there is no other choice.

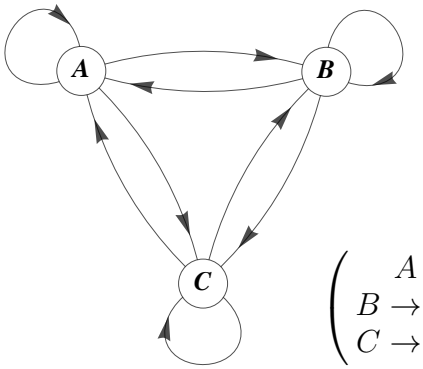
However their simple descriptions in English belie their relatively complex implementations in software — such as just saying "find cycles" and "merge." What are the implementation details, say of depth first search to find cycles, or the details of identifying bridges (especially when deleting edges changes the bridges)?

This common but deceptive simplicity of many published algorithms is a problem we also noted in our previous discussion of the Chinese Postman Tour [Thimbleby 2003a].

2.3. A new Euler cycle algorithm for complete directed graphs

Inspired by Hierholzer's algorithm, we make five observations:

(1) Instead of using an algorithm to find cycles, since the graph is complete we already know a decomposition into cycles: namely, for every vertex u there is a cycle of length 1, $u \to u$, and for every pair of vertices u, v $(u \neq v)$ there is a cycle of length 2, $u \to v \to u$. We call these *trivial cycles*.



$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} A \to A & \cdots & \cdots \\ B \to A \to B & B \to B & \cdots \\ C \to A \to C & C \to B \to C & C \to C \end{pmatrix}$$

Fig. 1. Representations of the complete directed graph K_3, where there is an edge $u \to v$ for each directed pair of the 3 vertices $u, v \in \{A, B, C\}$. As can be seen (left), the graph is composed from 6 trivial cycles: 3 single-arrow cycles (e.g., $A \to A$), and 3 double-arrow cycles (e.g., $A \to B \to A$). The corresponding cycle matrix is shown top right. The cycles represented by it are shown explicitly in the larger matrix, which has one entry shown explicitly for each trivial cycle. The omitted entries in the matrix are implied by symmetry, as, for example, the top right would be $A \to C \to A$ but this is the same cycle as $C \to A \to C$, which is shown bottom left.

(2) We can use a recursive algorithm to merge cycles. It starts walking any cycle, and if it crosses another cycle (that has not already been walked) it recursively walks that cycle, then resumes the cycle it was walking. This approach does not need any explicit operations or data structure to merge cycles.
(3) The recursive algorithm needs to distinguish the two types of trivial cycle, which depends simply on whether $u = v$.
(4) Once a cycle has been walked, it is not walked again. The algorithm therefore starts by initialising every trivial cycle as unwalked, and as it walks a cycle it marks it as walked, and hence will not walk it again.
(5) Marking trivial cycles can be represented by a Boolean matrix as follows: $walked_{uv}$ is true if the cycle $u \to v \to u$ has been marked as walked (if $u = v$ then the cycle $u \to u$ has been walked). We call this representation a *cycle matrix*.

Hence the following Euler cycle algorithm is suggested. Using C, vertices are numbered 0 to $N - 1$.

```
void cycle(int u, int v) // walk a cycle from u to u via v
{   if( !walked[u][v] )
    {   walked[u][v] = walked[v][u] = 1; // keep cycle matrix symmetric
        recordEdge(u, v); // record edge from u to v
        for( int w = 0; w < N; w++ )
            cycle(v, w); // unwalked cycles from v to v via w
        if( v != u ) recordEdge(v, u); // get back if not already at u
    }
}
```

As shown in figure 1, the cycle matrix is symmetric, and might therefore be represented more compactly as a triangular matrix. However, the algorithm implements `walked` as a square symmetric matrix (i.e., `walked[u][v]` = `walked[v][u]` is invariant) as this simplifies the implementation. It is possible to use a more sophisticated data structure than a matrix to avoid the `for`-loop and to avoid calling `cycle` when it will

immediately return: such an optimisation would not change the algorithm, but would significantly obscure its implementation.

The function `recordEdge` can be any way of recording the next edge along the Euler cycle; just printing it is easiest:

```
void recordEdge(int u, int v)
{   (void) printf("%d --> %d\n", u, v); // print an edge walked
}
```

The initial call of `cycle` to generate a solution is now simply:

```
cycle(0, 0); // Euler cycle starting at 0 returning to 0
```

which generates an Euler cycle that starts and ends at 0, with $0 \to 0$ as its first edge. We discuss alternatives in the next section, below, to randomise the cycle and to avoid always starting with the same edge.

Finally, the `walked` cycle matrix needs declaring and initialising:

```
int walked[N][N]; // represent cycles of a complete graph with N vertices

void initialise() // initialisation; all cycles initially unwalked
{  for( int u = 0; u < N; u++ )
      for( int v = 0; v < N; v++ )
         walked[u][v] = 0;
}
```

In C, arrays may be initialised to zero when declared, so explicit initialisation is not strictly required, but we provided explicit initialisation here in case the code needs to be re-entrant or is to be translated into another language.

2.4. Randomised sequences

Performing a sequence of experiments in random order controls the effects of drift and random error, and adding the constraint that every calibration value in the set must be preceded exactly once by every other calibration value in the set normalises any carry-over effects. Karl Popper called such sequences "shortest random-like sequences" [Popper 2002]. Therefore, for experimental design, we need to modify our algorithm so that the search for an unwalked cycle to recursively follow is randomised. The easiest way to do this is to use a random permutation in `cycle`'s `for`-loop, as follows:

```
void cycle(int u, int v) // follow a cycle from u to u via v
{   if( !walked[u][v] )
    {   walked[u][v] = walked[v][u] = 1; // keep the cycle matrix symmetric
        recordEdge(u, v); // record edge from u to v
        for( int w = 0; w < N; w++ )
            cycle(v, permutation[v][w]); // cycles from v via random vertex
        if( v != u ) recordEdge(v, u); // get back if not already at u
    }
}
```

There are N independent random permutations (indexed by `v`) to avoid dependencies between between vertices: `w` is mapped by `permutation[v][w]` to a random value in the range 0 to $N-1$.

The Knuth-Fisher-Yates shuffle [Knuth 1998b, p145–146] is a standard way to initialise such a permutation matrix:

```
int permutation[N][N];
```

```
...
for( int u = 0; u < N; u++ )
    {   for( int v = 0; v < N; v++ )
        {   int randomv = randInt(v+1);
            permutation[u][v] = permutation[u][randomv];
            permutation[u][randomv] = v;
        }
    }
```

To avoid always starting an Euler cycle from the same vertex and always starting with the same trivial cycle, the original base call `cycle(0, 0)` must be randomised too:

```
cycle(randInt(N), randInt(N));
```

Since the Knuth-Fisher-Yates shuffle is carefully designed to generate a uniform distribution of permutations, the choices made in the modified `cycle` will be uniformly random.

The function call `randInt(N)` provides a uniformly distributed integer from 0 to $N-1$ inclusive; it is not standard C, but can be approximated from the standard `rand()` function which produces a pseudo-random integer `0..RAND_MAX`.

```
int randInt(int n) // return random integer 0..n-1 inclusive
{  return floor(n*((double)rand())/((double)RAND_MAX));
}
```

This code assumes $N \ll \texttt{RAND\_MAX} \approx 2^{31}$ [Knuth 1998b, p119] — very reasonably, since the length of a Euler cycle is N^2, and it is hard to imagine experimenters (even robots) will have the time to perform sequential experiment runs longer than $10,000$, which assumption keeps $N \leq 100$.

Finally, to ensure different random sequences each time the program is run, the random number generator must be seeded differently during initialisation. This may be done in C from the current time:

```
time_t t;
srand((unsigned) time(&t));
```

2.5. Solving the wine tasting problem

The basic algorithm, described above, uses integers as the names of vertices. For many applications it may be more appropriate to use strings. We can define vertex name strings:[4]

```
// assuming N = 3 (C permits N > 3, which will not work well!)
char *wines[N] = { "Merlot", "Pinotage", "Shiraz" };
```

and then convert the edge recording by changing `recordEdge` to print the names of vertices rather than their numbers:

```
void recordEdge(int u, int v)
{   (void) printf("%s $\\rightarrow$\n", wines[u]);
}
```

... and finishing with a final `printf("%s\n", wines[0])`, which was otherwise lost in changing the `recordEdge` to only print one vertex rather than pairs. Note how we

[4]Not shown here, but our wine sequence generating program, which is generated automatically from this paper using our tool **relit** (see section 3.1), checks at run time that `N` correctly matches the number of wines declared. (It is a shame that this cannot be performed with a static check in C.)

used LaTeX's `$\rightarrow$` to generate a "$\rightarrow$" symbol to make the experiment sequence look a bit neater when typeset by LaTeX. With these changes, the code generates the following sequence: Shiraz $\rightarrow$ Pinotage $\rightarrow$ Merlot $\rightarrow$ Shiraz $\rightarrow$ Shiraz $\rightarrow$ Merlot $\rightarrow$ Merlot $\rightarrow$ Pinotage $\rightarrow$ Pinotage $\rightarrow$ Shiraz.

It is easy to confirm by hand that this is indeed an Euler cycle (perhaps by mapping wines to the letters A, B, C and ticking off the edges in figure 1). Note that, as an Eulerian cycle ends at the starting vertex, an additional drink of Shiraz (in this worked example)[5] is required: thus a balanced scientific experiment would repeat sequential experiments with randomly selected starting vertices each time, using the ideas of section 2.4. Randomisation avoids the potential bias caused by drinking too much Shiraz — in the sequence above Shiraz happens to be drunk both first and last, and hence once more than any other wine. Randomisation also avoids the possible bias caused by always starting with Shiraz, which might, for instance, be somebody's favourite wine. After enough randomised sequential experiments, these biases will be evened out.

Wine buffs will be tempted to test the algorithm with larger N. It would certainly be fun to take four varieties of wine to Königsberg (the original graph had 4 vertices and 7 edges), cross a bridge, drink the wine variety available on the land, then cross the next bridge, and consider the wine on the other side of the river. As it happens, Königsberg does not have an Euler cycle, so this experiment could take a long time and might result in a new meaning for "drunken walks."

3. FINDING ALGORITHMS

While we continue to take it for granted that algorithms are described in English, in psuedo-code, with illustrative fragments of uncompilable code, or left as exercises for the reader, unnecessary errors often persist in their real-life implementations. (Some of the problems are reviewed in [Thimbleby 2004; 2012].) Often, of course, publications are not always aiming to describe the algorithm as such, but to do something else — like teach students, analyse complexity, prove some theorems, discuss how to optimise them, and so on. This "dual use" creates things that look very much *like* algorithms but which cannot be reliably used in practice *as* algorithms; the likely confusion calls to mind Feynman's critique of cargo cult science [Feynman 1992].

Ironically, then, in one of the very few areas — programming — where we could be completely explicit about our work (e.g., if an algorithm works, it is text that can surely be published explicitly) there is a default culture of vagueness and "abstraction" that undermines scientific reproducibility if not progress. English, pseudo-code, fragments, exercises ... none are scientific statements: they are irrefutable [Popper 2002].

Often software practice and experience emphasises the efficiency of running programs, but we should also be concerned with the end-to-end time to develop a program, confirm it is correctly implemented, and to run it. In many cases, the development time dominates the run time. Unlike the bulk of the algorithms textbook literature, our goal was to have a reliable program quicker, not a faster program later. As we shall argue, literate programming and its variants are powerful approaches to be more scientific when publishing algorithms, and hence to end up with more reliable programs working in the wider world. In this paper, we developed a new variant of literate programming for this very reason.

[5] The wine experiment sequence was generated automatically from the randomised version of the code shown in this paper; it was run and the results saved to a file `winelist.tex`, the last line of which was extracted (several times) for this paragraph by using Unix's `tail -n 1 winelist.tex`. Each run of the program will generate a different random example for this paper.

Programs that can be compiled and run involve details that are generally not relevant to discussions of algorithms. Furthermore, writing a paper or book is a human process, so transcription errors may creep into any algorithms presented.

There is even the danger that publications may be sloppy: sometimes, authors do not check their published code adequately and referees take the correctness of the code on faith (partly because it is too hard to reconstruct the code from the paper, and too hard to disentangle whether problems are due to the published code or the referee's own errors in the reconstruction of it). "Sloppy" is a harsh word, but it covers a wide range of common problems ranging from deliberate fraud, unintentional exagerration, accidental uncorrected errors, and well-intentioned aspirational comments — like, the program would *obviously* work like this (even if it doesn't quite work yet). Even trivial and excusable errors, like typos and spelling errors, undermine the reproducibility of program code.

All this means that finding an algorithm in the literature that can be used to solve a real problem is fraught with difficulties. In our case, having developed an algorithm to solve our problem — because it is a very common problem for experimenters who may not have sufficient programmering expertise — we wanted to take care that what we published (i.e., this paper) would be both clear and able to be copied from the paper as real, executable code without problem. We wanted to make it readily — and reliably — available.

At the start, we developed and wrote our algorithm first in Mathematica then in C, which is non-proprietary and more portable. Mathematica is excellent for presenting the *results* of programs in papers, but it is not good for writing papers *about* algorithms, as it requires code to be runnable. In a paper you generally want to discuss fragments of an algorithm in any order that suits the exposition. But because we were excited by our algorithm, we started to write this manuscript, using the very flexible typesetting system LaTeX [Lamport 1994] to publish and share our findings.

As we wrote, we reviewed and revised the paper. We naturally made many changes to the program code. For example, originally we had used variables `i`, `j`... in the program, but for writing the paper we decided to use `u`, `v`... as these are conventional names for graph vertices.[6] So, over time, the original code and its description in the manuscript drifted far apart; yet, in principle, it should have been essential that the manuscript was synchronised with the source code so that it described the source code without error. Ideally, the paper should automatically change to reflect updates to the source code.

In a word, we did not want to be sloppy, yet our initial approach to writing the paper was making life hard. It was tempting to take an easy approach, and only describe our algorithm in words or pseudo-code, being a bit vague about the details. Conventionally, neither the readers of the paper nor we, the authors, would worry about slight discrepancies, because they would be invisible and unknown.

An obvious way to help ensure that executable code transcribed from a manuscript is correct is to cut and paste the relevant text in the paper to reconstruct a new program. However the presence of typesetting commands in LaTeX documents means that the LaTeX code used to generate the manuscript will by necessity differ from the source code of the executable file, so a simple "cut and paste" approach is unreliable. But why do something by hand when you can design a tool to do it with far more generality and reliability? Once a tool is written to do this chore automatically, this frees the authors from worrying about maintaining and checking the code in the document as it is repeatedly edited and revised: it should be done automatically.

[6] Unfortunately wine names starting with u, v, w, are not commonly recognisable.

This idea is very similar to Knuth's *literate programming* [Knuth 1992] which combines source code with explanatory documentation; however the format of the documentation generated by literate programming is not suitable for a journal manuscript. Literate programming also has the disadvantage for us (and for many authors) that the author has to start with the literate program, and in this instance we had already drafted the manuscript as a LaTeX document. **Warp** is a type of literate programming that extracts code from a normal program commented in XML, thus avoiding the separate processing that literate programming normally requires to generate the executable program [Thimbleby 2003b]. Our earlier paper on the related Chinese Postman Tour [Thimbleby 2003a] used **warp** to present accurate Java code.[7] **Loom** [Hanson 1987] (originally written by Janet Incerpi and Robert Sedgewick for their classic algorithms book [Sedgewick 1983]) is another approach, similar to **warp**, but allows the use of Unix filters to perform arbitrary transformations of code (e.g., to handle special symbols) that is then inserted into arbitrary documents.

In the present paper, however, we had already been working on the code *in* the paper, in the usual informal way. To avoid this becoming increasingly sloppy (or, conversely, a huge burden to manage), we therefore developed a novel "reversed" literate programming approach: using it, the code is exactly as written in this paper (i.e., what you are now reading) *and* it can be automatically extracted to generate a program that is directly executable. The point is: we know that the code shown in this paper works, and moreover, we have a lightweight, fully automatic process that goes directly from this paper to executable code. What you now see may not be all of the code,[8] but the code shown *does work* as shown.

Figure 2 compares the main forms of literate programming, including our new approach. There are of course many other related approaches, ranging from the very simple such as our reverse literate programming to the highly sophisticated, such as PreTeX [Kruse 1999] that are designed for large complex projects and have commensurate learning curves: for a review see [Thimbleby 2003b] and [Wikipedia 2016] for an up-to-date summary of available tools.

3.1. Reverse literate programming — the details

Code formatted in this paper is written in the following conventional LaTeX style:

```
\begin{verbatim}
(void) printf("Hello reverse literate programming!\n");
\end{verbatim}
```

There is nothing unusual in this approach. However, we need to be able to generate a compilable program from such code snippets, even though they may be scattered in any order throughout the paper. In our approach, using our new tool **relit**, we name these snippets, and then use the names to generate compilable code.

A name is defined by preceding any part of the LaTeX document with a special comment:

```
%define name /start pattern/±offset, /end pattern/±offset
```

The pattern style (general regular expressions are permitted) is deliberately reminiscent of the form used by the Unix utility **ed**, and the entire line (starting with the

[7] The Chinese Postman finds the shortest cycle in a weighted graph that is not necessarily Eulerian (i.e., some edges may need to be walked more than once); it is a non-trivial generalisation of the problem discussed in this paper.

[8] There is exactly one line missing; see section 3.3.

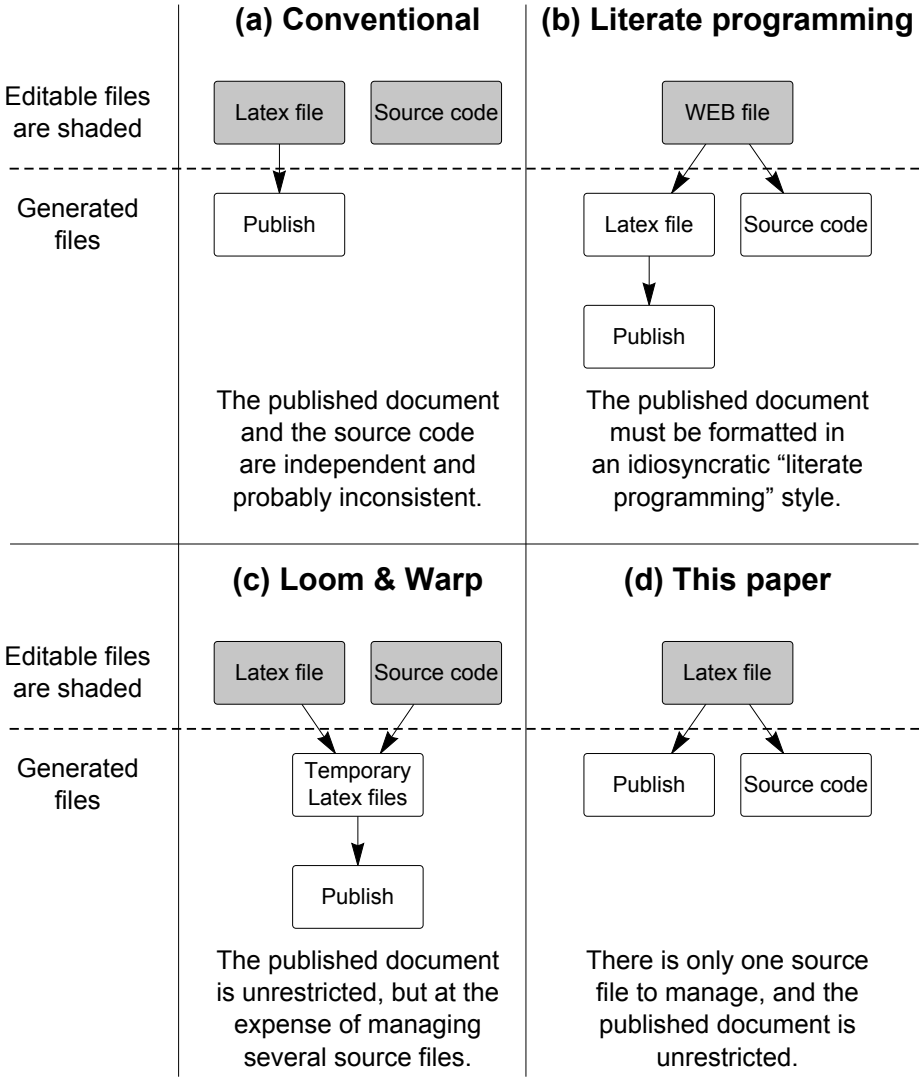


Fig. 2. Comparing various ways to write about programming. Note that in the conventional approach (a) there is no guarantee that the published paper faithfully represents the source code, as the paper and source code can be (and will be) edited independently: what is published has no automatic connection to the source code. (Although not made clear in the diagram, typically source code and LaTeX documents will be split into multiple files for convenience. All methods can handle multiple files.)

standard LaTeX comment symbol %) is comment so it is ignored completely by LaTeX itself. The effect is that *name* is defined to be the text in the file over the specified range of lines. For example, the code extract above could be preceded with a definition of the name `demo`:

```
%define demo /verbatim/+1, /verbatim/-1
\begin{verbatim}
(void) printf("Hello reverse literate programming!\n");
```

```
\end{verbatim}
```

This defines the name `demo` to be the text between the two `verbatim` lines (which are required by LaTeX to typeset the code to see as shown above), namely `printf("Hello reverse literate programming!\n");`.

A document can generate any number of source code files, by writing analogous file definitions such as:

```
%generate filename ., /%end/-1
text...
%end
```

where within *text* any occurrence of <*name*> is replaced by its definition, and so on recursively. For example,

```
%generate hello.c ., /%end/-1
int main(int argc, const char *argv[])
{ <demo>
  return 0;
}
%end
```

will generate a file called `hello.c` that should compile and say hello — except, of course, compiling it will generate the warning that `printf` has not been defined. But that is the point: the reverse literate programming enables us to write a paper *and* check whether the code we are writing is valid.

Illustrating **relit**'s commands out of context (i.e., not showing the surrounding English descriptions of the code), as above, does not show the powerful leverage the approach provides for reflection. The description and explanation of code embeds the code; it is thus easy to improve the description *and* easily improve the code, and conversely. The author can think in either language, and slip in and out of either description without losing track of their thoughts.

As a matter of fact, we realised we needed to add some explicit code to the paper only after we had implemented reverse literate programming and found some oversights in our original description. Reverse literate programming helped us find the problems: various declarations and a bit of wider context were needed to make the code described in this paper compile and run without error. The following code generation, creating a file `euler.c`, works correctly once **relit** substitutes the values for the various names, which have been defined elsewhere in this paper:[9]

```
%generate euler.c ., /%end/-1
#define N 3 // for a graph with N vertices
<common-declarations>
<declare-walked>
<define-basic-recordEdge>
<define-non-randomised-cycle>

int main(int argc, const char *argv[])
{ <main-body>
  return 0;
}
%end
```

[9]The code shown here *is* the actual code that **relit** used — we just placed the **relit** % comment inside the LaTeX `verbatim` environment so it could be seen.

This code was written exactly as shown, and it is easy for the authors to check it actually works: with **relit** it generates the file `euler.c` which can be compiled and run, and will get the expected results. This implies the code `<main-body>` and so on recursively also compiles and works. Using **make** [Mecklenburg 2004] with a makefile (also generated by **relit** from the same document) made it very easy to automatically update the code files and unit test them repeatedly as the publication evolved.[10]

Of course it is unlikely that a normal paper (that is, one not explaining **relit**) would present the code shown explicitly above: usually, the `%generate` command would have appeared after LaTeX's `\end{document}` so it would disappear and not be typeset as part of the published paper. Similarly, any details that need not be part of a typical published paper can be hidden. As we will discuss in section 3.3, **relit** can tell the author what code is visible and what code has been hidden to avoid any uncertainty.

Notice that **relit** allows the generated program code to be put in the correct order required for a compiler, but the visible code it is assembled from — here, `<declare-walked>`, etc — can be presented to the reader in whatever order is best for the narrative of the publication. For example, although `<main-body>` appears near the end of the code, it might be explained to the reader before showing increasing levels of detail of the implementation.

Name definitions can occur freely before or after they are used. For example, in this paper the name `<common-declarations>` (used above) is defined after the end of the LaTeX document: the headers are a detail we feel readers of the present paper do not need to see written out in detail, but they are necessary to be able to generate executable code. There are no restrictions on the order of generating files: they, too, can be specified in any order that best suits the needs of the author.

This complete freedom of order distinguishes **relit** from typical "notebook" approaches such as used in Jupyter [Project Jupyter team 2016] and Mathematica [Wolfram 1999], which both impose a sequential order on the document to suit the compiler rather than the reader or author. Notebooks also require units of code to be syntactically complete, which is not required by **relit** either, as made clear with the Knuth-Fisher-Yates shuffle example in section 2.4.)

As well as in the document, as shown above, names can also be defined in **relit**'s command line parameters, so any textual information can be imported when the tool is run — such as a version number, the date, or even output from **expect** to refer to unit test diagnostics.

Our tool **relit** provides all the normal checks, such as reporting if names are defined and never used, used recursively, multiply defined, etc. Indeed, some authors have used deliberate "errors" as a way of providing metacomments: defining an unused name results in its name and value being reported to the user — so the text is highlighted, which can be used as a reminder to fix an issue with the document.

Relit is a Unix tool, written as a short C program, 380 lines long and, together with this paper and documentation, is available from https://github.com/haroldthimbleby/relit.

3.2. Generating any text, not just program code

In general, authors of papers may want arbitrary text generated, not just programs (for example, we showed the text generated by running our algorithm on a selection of wines in section 2.5), and the generated text may require sophisticated processing or testing.

[10] By default **relit** only updates files when their content changes, which makes using **make** efficient, as most edits to a document do not update any files generated from it.

There are many ways to do this: **loom** for example, generalises LaTeX's `\input` command to allow arbitrary processing, but this has the disadvantage that the approach requires an intermediate LaTeX file to be generated. Instead, our simple approach is to use `%generate` to create a makefile or any shell script: then any processing whatsoever can be performed, and of course it will typically generate files that are then included in the LaTeX paper. Indeed, this is how we generated the sample list of wine tasting — using the makefile generated by **relit**, Unix's **make** then generated a C source program `wine.c` from the paper you are reading, which was then compiled and run (in the same run of **make**), obtaining results saved to a file. Finally, that file was read in at the appropriate point when this paper was typeset (in section 2.5), using `\input` to read in the program's saved output to insert it into the paragraph where it was needed.

Of course, files created by **relit** programming can be processed by other tools in arbitrary ways, for example stream editors to decode LaTeX typesetting conventions (e.g., in normal LaTeX, the common programming symbol `&` has to be written `\&`).

3.3. Checking hidden material

The authors of a paper should be able to check that the explicit code shown in their paper is exactly what they want the reader to see and that nothing critical has been omitted. On the other hand, a published paper should conceal details that are distracting or irrelevant to its core message: it need not show all of the code a runnable program requires, but just enough to get the idea across. The tension between these, being explicit and being concise, is a recipe for error: what is hidden, by definition, cannot be seen, yet some omitted information may be required for achieving a complete program. Worse, the reader of the paper may not be certain what is missing, and they may not have the skills or time needed to reconstruct it correctly. Compounding the problem is the so-called "curse of knowledge" [Pinker 2015]: the authors of the paper have privileged knowledge (in principle they know everything about what they are talking about) and they may therefore be unaware that some things have not been explicitly mentioned in the paper — it is very hard to distinguish between what they know in general about what they are writing and what they think they know (perhaps inaccurately) is in the paper.

Tools like **warp** and **loom** help the authors ensure that code published has been obtained directly the working programs, but it is still possible to write code in the paper that has never been tested or compiled, and also to leave a lot of essential context in the program that **warp** or **loom** do not draw into the document. Figure 3 summarises the challenges.

Unfortunately, in reverse literate programming, since generated code can include names that are defined anywhere in the documents, and the defined values themselves may or may not be visible in the published paper, and so on recursively, it is impractical to manually determine what code is visible in the published paper and what, if any, is not visible.

Our reverse literate programming tool, **relit**, allows code to be tagged, and although the tags can be used for any purpose, a useful application is to keep track of what code is visible and what is not. The approach is simple; the `define` and `generate` commands can be followed by optional tags:

```
%define name start, end [, tag]
%generate filename start, end [, tag]
```

For every file generated, **relit** additionally generates a duplicate file but marked up with the tags, with the relevant tags output as each *name* (or *filename*) is expanded. The tags are arbitrary text, but will typically be LaTeX macro names that can be defined to highlight text. Of course if the tags include names like <*stuff*> then they will be

Reverse literate programming	A negligible amount of code may be hidden, but only with deliberate effort. All code remains in the same file — so it is easy to see and edit without introducing errors. Negligible learning curve; no extra steps. Intended for peer reviewed publications.	⋆⋆⋆⋆ Typically only one file to maintain.
Loom and Warp	Some code is visible, but most is hidden — it remains in separate code files. Small learning curve; extra step needed in writing process. Good for publications.	⋆⋆⋆ Any number of files.
Literate programming	No code is hidden and all the code and document is in one place, but there is usually so much code that not all can be reliably read by a human. Steep learning curve; extra steps needed in writing and compiling processes. Ideal for internal documentation rather than peer reviewed publications.	⋆⋆ Typically few files to maintain.
Conventional approach	Source code is hidden. There is no link between the source code and the published paper. Errors and inconsistencies are easy to introduce. No learning curve.	⋆ Any number of files.

⋆⋆⋆⋆	All source code and published paper in a single document. Diagnostics for both missing and defined but unused code. Flexible indexing, in published document and in command line.
⋆⋆⋆	Some source code is published. Diagnostics for missing code, but some code can be defined (i.e., intended for use) but accidentally left unused. No indexing.
⋆⋆	All source code is published. No diagnostics necessary. Sophisticated indexing in published document.
⋆	No connection between published paper and program source code. No diagnostics possible. No indexing.

Fig. 3. The more code that is hidden, the more likely it will drift into complexity and concealed critical details. Conventional literate programming hides nothing, but typically makes the result too large to be publishable (**JavaDoc** only documents the API rather than the entire program). **Loom** and **Warp** help ensure code published is correct, but it may be incomplete. The present paper's reverse approach, as embodied by **relit**, ensures almost all the code is published, and what the authors may choose not to publish can easily be checked with simple diagnostics (see section 3.3) — and what is not published remains in the original LaTeX files, so the authors always remain aware of it.

expanded as normal (the names can be defined anywhere, generally after the end of the document). This feature is useful if the tag is complicated (e.g., writing `<tag>` is easier and more reliable than writing out a tag in full every time it is needed) or if the author wants a tag to have many lines of text.

With tagging, we can readily obtain typeset text showing where code has come from.

However, since tagging each *name* definition is a bit tedious — and therefore itself error-prone — a default tag can be defined:

```
%set-tag tag
```

That *tag* is then automatically applied to all subsequent definitions (and files) until it is superceded, or overridden by explicit tags. Hence, typically a LaTeX document will start:

```
...
\begin{document}
%set-tag \seen{}
```

```
...
published document including visible definitions
```

and then have the following at its end:

```
...
\end{document}
%set-tag \unseen{}
...
hidden definitions and files
```

Here, the LaTeX code `\end{document}` signals the end of the published LaTeX document, and all subsequent text will be hidden from view in the published document. Normally this part of a LaTeX document is empty, or has accummulated "junk" text and thoughts the authors cannot steal themselves to *really* delete — the space has a useful role in co-authored documents, where one author wants to delete text from the published document, but does not want another author to lose some idea without a chance to reconsider it before it is deleted. In our case, with reverse literate programming, the hidden space can also be used for defining program code that is needed for compiling and testing, but is considered too much detail for visible inclusion in the published document.

The illustrative tags `\seen{}` and `\unseen{}` used above are arbitrary; one might choose to use `\color{black}` and `\color{red}` instead, say. In the complete example shown below (having defined `\unseen{}` appropriately) the "`*** hidden ***`" marker is provided automatically and therefore correctly.[11]

The code shown next, below, is one of the example algorithms as already discussed in this paper: the highlighted text generated by **relit** reveals code that was not previously shown in the paper but which was generated for the compiled test programs. In other words, the highlighted code is included in this document, but it was hidden from sight from readers of this paper by being placed after `\end{document}`.

The authors of this paper are happy that these standard declarations are not taking up space in the published paper — in any case, if a reader of this paper faithfully copies the published code and omits these lines, good compilers will provide helpful error messages such as "Note: include the header `<stdio.h>` or explicitly provide a declaration for `printf`."

```
#define N 3 // for a graph with N vertices
#include <stdio.h>                                          *** hidden ***
int walked[N][N]; // represent cycles of a complete graph with N vertices
void recordEdge(int u, int v)
{   (void) printf("%d --> %d\n", u, v); // print an edge walked
}
void cycle(int u, int v) // walk a cycle from u to u via v
{   if( !walked[u][v] )
    {   walked[u][v] = walked[v][u] = 1; // keep cycle matrix symmetric
        recordEdge(u, v); // record edge from u to v
        for( int w = 0; w < N; w++ )
            cycle(v, w); // unwalked cycles from v to v via w
        if( v != u ) recordEdge(v, u); // get back if not already at u
    }
}
int main(int argc, const char *argv[])
```

[11] Correctness here depends on the author not cheating! LaTeX is programmable, so a determined author could defeat the reliability of the tagging mechanism if they were so inclined.

Normal mode	**TEX-mode**
% define *name* re_1, re_2	\relit{define *name* re_1, re_2}
% define *name* re_1, re_2, *tag*	\relit[*tag*]{define *name* re_1, re_2}
% generate *name* re_1, re_2	\relit{generate *name* re_1, re_2}
% generate *name* re_1, re_2, *tag*	\relit[*tag*]{generate *name* re_1, re_2}
% set-tag *tag*	\relit{set-tag *tag*}
% *any comments permitted*	\relit{ends}

Fig. 4. Corresponding normal and TEX-mode syntax for **relit** commands.

```
{ cycle(0, 0); // Euler cycle starting at 0 returning to 0
  return 0;
}
```

To summarise: using this feature of **relit** we can assure ourselves that readers of this paper have the complete algorithm to reproduce exactly or as exactly as we want. The one line, highlighted above in the diagnostic output from **relit**, shows that in fact our paper does not disclose one line of code. However we consider this omission obvious, unnecessary and distracting implementation detail for the paper.

Apart from the original literate programming approach that tells the reader everything, which may be overwhelming, we know of no other approach that give the benefits of concise algorithm publication combined with such assurances of reproducibility.

3.4. A more flexible TEX-mode

As described above, **relit** uses % to introduce commands, like `generate` and `define`; the advantage of this approach is that LATEX completely ignores **relit** commands because they are in the form of LATEX comments. However, in addition, **relit** provides an integrated "TEX-mode" that makes the syntax it uses real TEX or LATEX code; this is more sophisticated, requires a little TEX programming, but has advantages that some authors may appreciate.

Comments, text after %, get completely lost to TEX and LATEX, which makes the basic use of **relit** very easy to understand: **relit** has nothing to do with how they work. On the other hand, an author may wish to see exactly where the **relit** commands are and what they do, which means the commands should not be ignored by the typesetting programs. In **relit**'s "TEX-mode," therefore, commands are written as proper TEX commands rather than as comments.

TEX has to define the **relit** commands that are used in TEX-mode: for example, if we want **relit**'s behaviour to be exactly as described above then TEX must define **relit** commands so they are ignored. However, TEX can also be used to define the commands to do more interesting things, such as printing where they are used in the document to help the authors prepare the paper. TEX can also construct an index of **relit** names, which may help in large authoring projects. An example is shown in figure 5.

The TEX-mode syntax corresponds directly to the original syntax, and is summarised in figure 4. The final case in the table allows the author to use `\relit{ends}` to mark the end of definitions (normally anything convenient, such as text in a comment, can be used). Here is a (contrived) example of its use:

RELIT * RELIT: generate TeX-mode-demo.tex /indent/, /relit.ends/-1, \seen{}

This and the next paragraph together make up a simple example that generates a file `TeX-mode-demo.tex` that will contain this text after **relit** is run.

Although normally ignored by LATEX and invisible to readers, the **relit** commands used to save these two paragraphs have been made clearly visible by defining `\relit` appropriately in LATEX, as can be seen highlighted on the two lines just before and just after these two paragraphs.

Fig. 5. An extract from a simple **relit** TEX-mode index. Only names and files explicitly shown in this paper have been included (i.e., definitions where the normal **relit** process has not shown names to the reader, along with all hidden material after the \end{document}, have been excluded).*

```
* RELIT: ends
```

RELIT

Notice in this example how we have defined \relit so it typesets its parameters to appear directly in the typeset document; this approach can be very useful for helping rewriting and refactoring. The \relit definitions used in this paper are readily available: in the same way that **relit** is run on this paper to generate compilable programs, an additional %generate command simply saved the actual definition of \relit used above to a file so it can be reused in other documents.

Relit warns if TEX-mode and non-TEX-mode commands are mixed in the same files: the point of TEX-mode is that authors can use TEX to keep track of *all* **relit** commands, but using the % **relit** notation as well (which of course TEX completely ignores) would undermine reliable tracking.

The reason that **relit** makes TEX-mode an option, rather than being the only method of use, is that the feature introduces a conceptual layer of complexity the standard approach does not require. Using TEX-mode requires not just defining some TEX commands (a matter of copying them from somewhere that already works) but also understanding how to balance TEX's syntax with regular expressions, which is harder as you can no longer use \, %, {, }, etc, in the same way.

3.5. Thoughts on further work

The experience of developing and using a tool drives new ideas, creating a tension between polishing the legacy or generalising and extending the scope and reach of the ideas. Here are several key potential developments:

(1) The syntax of **relit** could be improved. For example, instead of defining names or generating files, **relit** commands could be considered as specifying text from an expression, which could then be processed. Like **loom**, commands could pipe their output to files or other Unix processes. This would generalise **relit**. At present, the same effect can be achieved, albeit after one extra step, by using makefiles and other external processes filtering the files that are generated — but this is

*This paper, which is both a paper explaining an algorithm in the usual way and a paper explaining **relit** in detail, is complicated by unusually having to write **relit** commands *inside* LATEX verbatim environments, so the reader can see them but where simultaneous indexing is impossible. (In normal use, **relit** commands would be outside verbatim and similar commands and hence they would be processed by LATEX and would normally be invisible to the reader.) Thus, index entries marked * in figure 5 were provided by using duplicated **relit** commands written just outside the verbatim environments so LATEX could evaluate them to make the starred index entries as shown.

"bad practice" as it involves creating an arbitrary namespace, specifically the names of the temporary files.

(2) **Relit**'s syntax is fixed, and needs internationalising and making appropriate for languages other than LaTeX. At present, **relit** is open source and available for any such improvements.

(3) **Relit** is an example of a "parallel language": a language grafted into an existing language, in this case, **relit**'s simple commands squeezed into LaTeX. Parallel languages are very common, but all of them are *ad hoc* and are compromises. Think of HTML, CSS, PHP and JavaScript that have developed conventions so that they can co-exist; or Java and JavaDoc; C and its `#define` macro-language which defies C's own syntax; or XML and its metadata; even LaTeX and TeX. Conversely, JSON has no parallel language. More research is needed on parallel languages, so that when new languages are introduced they can be extended in the future without unnecessary and perhaps fragile compromises.

(4) **Relit** could be extended with its own parallel languages. **Relit** allows LaTeX and code to be interleaved, but an author may want additional information or annotations, such as program specifications, test cases, invariants, contracts, author names, versioning information, etc: these might be neither program code the author wants the reader to see, nor program code that should be compiled.

(5) Finally, in this paper we have argued the value of tools like **relit**. For good reasons, we believe it enormously helps reproducibility and dependability of publications. Whether it *actually* helps other scientists, and whether it can be improved to help more authors to achieve their publication goals, is an open question — a more profound question, which is harder but more useful to answer is: of several **relit**-like systems, which are more effective and why? We currently lack the theories and principles of doing and disseminating computer science so these critical questions are hard.[13] Our final section, next, explores these bigger issues.

4. THE BENEFITS OF AUTOMATIC HONESTY

Richard Feynman's opening sentence for his 1965 Physics Nobel Prize lecture (shared with Sin-Itiro Tomonaga and Julian Schwinger) is

> "We have a habit in writing articles published in scientific journals to make the work as finished as possible, to cover all the tracks, to not worry about the blind alleys or to describe how you had the wrong idea first, and so on."
>
> [Feynman 1965]

Here Feynman recalls the common habit we are not immune to in Computer Science when writing articles about algorithms (and when using other formal languages). As we refine and improve our articles, their connection with the orginal programs becomes tenuous. We ourselves encountered this problem in very early drafts of this paper before we had invented **relit**: improving the paper led us to talk about variables u, v yet the program we were writing about really had variables i, j. Our original programs had many details, including essential declarations, that never made it into the working paper. It is too easy to slip from clarifying your writing to simply making it up.

There is a legitimate stage in drafting papers where authors can freely write about what they hope and intend will be true as placeholders for future work. "Our program *will* work like this ..." But if what they write has no rigorous connection with reality

[13] It is noteworthy that the ACM Computing Classification System (CCS, dl.acm.org/ccs) does not classify the activity that defines our discipline: publishing.

it will never be straightforward to know when the aspirational gap has been closed, and the authors thus risk eventually publishing misleading and time-wasting papers. Readers generally have no way to distinguish sketches and visionary ideas from actual achievements.

Feynman later warned about the utter honesty essential to do good science. In 1974 he said,

> "It's a kind of scientific integrity, a principle of scientific thought that corresponds to a kind of utter honesty — a kind of leaning over backwards. [. . .] In summary, the idea is to try to give all of the information to help others to judge the value of your contribution; not just the information that leads to judgment in one particular direction or another." [Feynman 1992]

But with computers we can go further than physicists: we can help make honesty automatic. This is so important: when honesty is automatic, it is easy *and reliable*. As Feynman said,

> "The first principle is that you must not fool yourself — and you are the easiest person to fool. So you have to be very careful about that. After you've not fooled yourself, it's easy not to fool other scientists. You just have to be honest in a conventional way after that."

So automatic tools and techniques to encourage reproducibility, like **relit**, can help you help yourself — and help your readers. Your readers should not have to work out things for themselves you did not know you were not telling them. Our scientific writing should not rely on readers having to use their insight to interpret and clarify what we write; different readers may have different insights, and then it is not clear what our science is communicating.

When describing something that is complicated, there is a temptation to simplify and take short cuts. In the worst case, this results in publications that describe what ought to happen, what we hope happens, but there are omissions that mean the claims are not easily reproducible. Somehow it is too easy to reframe our programs so that it sounds as if they work; all the details are too hard to check even for the original authors — and as we convince themselves what we write is correct, then there seems to be less and less need to go to the trouble of checking our code. When we conceal errors — and, worse, when we conceal errors from ourselves — although our work may look good, it holds back progress [Syed 2015].

If a tool like **relit** is used, the code in the paper is exactly what works. If describing it gets complicated, this cannot be denied. Instead of simplifying the narrative, the author is encouraged to improve the code so it becomes easier to describe. This means *both* the code and the paper improve, and improve together.

It must be pointed out that not everybody agrees under all circumstances. For example, Knuth writes:

> "The author feels that this technique of deliberate lying will actually make it easier for you to learn the ideas. Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions." [Knuth 1986, p*vii*]

His argument is that a reader learns, and at first things need to be simplified with "white lies" as he calls them [Knuth 1986, p44], and then, later with more knowledge and skill, the reader can learn from more elaborate explanations that would have been incomprehensible earlier. However, this is Knuth's strategy for writing his substantial and very successful *TEXbook*, which describes in a single document a very complex system, namely TEX itself. He knows readers will have an arduous process ahead of them, and the book tries to fit the needs of both beginners and experts. We believe that

in contrast, in normal scientific publishing, particularly in publishing comparatively short peer reviewed papers — which normally focus on one or two ideas — the utter honesty championed by Feynman has important if not overriding advantages. Another point of view is that perhaps Knuth should have improved TEX so it did not require lying to explain it so well. On the other hand, one of the enormous strengths of TEX is that it has *not* "improved" over time, so it is remarkably portable and dependable, unlike many programs that are continually "improving" with new changes forcing users to upgrade.

While the *TEXbook* is an exceptional piece of writing, elsewhere Knuth himself has said science is what we can explain well enough to computers [Petkovšek et al. 1996], and that everything else is art.

Our position is that if people are publishing scientific papers about algorithms that cannot "be explained to computers" — they surely ought to work as described — then their papers are too obscure to be called science. They are compromising reproducibility. Science advances when art becomes science; we hope, then, that ideas like reverse literate programming will help move the darker arts of publishing programming into an improving science of programming.

In practice, Feynman's radical integrity is discouragingly hard work unless tools help make it automatic. Here is an example of automatic integrity achieved with the help of **relit**. At the last moment, our colleague Paul Cairns pointed out that we had confused wine regions (e.g., Chianti) for grape varieties (e.g., Pinotage and Merlot). In the conventional approach to writing about algorithms, we would have edited the paper ... and the paper and the programs we had written about would have diverged, or at least we would have needed to edit several files that would then have required tedious manual cross-checking (if we could be bothered to do it). We might have decided it was not worth the bother. We might have fooled ourselves: the program works even if it might get the names of the wine regions or grapes mixed up. Certainly, explaining what needs doing is easier than doing it, and it is then but a short step to justifying to ourselves not doing it because it is easier imagined than done.

Instead, because we were using **relit**, we did *one very trivial* global edit *in a single file* which immediately replaced all our original uses of the word Chianti with Shiraz. This was a trivial change. With no further effort, thanks to using **relit**, we automatically had a new C program that used Shiraz, and the output from running it was then automatically inserted back into this paper so the example output (used in section 2.5) was also updated to say Shiraz instead of Chianti. Everything remained consistent with no effort. And everything is obviously consistent with the code shown in the paper because it was generated by running exactly the code in the paper that you are reading. Doing it is easier than explaining it.

5. CONCLUSIONS

We developed an elegant algorithm to help perform reliable sequential experiments, and in the process of writing about it clearly we also developed a tool to help publish reliable papers about algorithms. This paper combined a description of the algorithm and a description of the tool that enabled us to write this paper.

Our algorithm generates Euler cycles or randomised Euler cycles for complete directed graphs. It can be used to generate randomised $(N, 2)$ de Bruijn sequences for efficient sequential experimental design which controls for random variation, systematic error, and carry-over effects. The simplicity of our algorithm, written in basic C, means that it is easy to understand and translate into other languages, which makes it accessible to those engaged in experimental research.

Conventional literate programming prevents transcription errors by linking source code and manuscripts. Our novel "reverse literate programming" approach, which we

have described and used in the preparation of the present paper, has allowed us to simultaneously edit the program code and its description in this paper at will, and repeatedly automatically generate an executable program we could run and use to confirm the integrity of the code as described exactly in the paper.

This paper and the open source **relit** tool and documentation are available from https://github.com/haroldthimbleby/relit

Acknowledgements

We are very grateful to Paul Cairns, Rod Chapman and Bob Laramee for helping us greatly improve the paper.

REFERENCES

G. K. Aguirre, M. G. Mattar, and L. Magis-Weinberg. 2011. de Bruijn cycles for neural decoding. *Neuroimage* 56, 3 (2011), 1293–1300. DOI:10.1016/j.neuroimage.2011.02.005

T. H. Cormen. 2016. Thomas H. Cormen Professor Department of Computer Science. *http://www.cs.dartmouth.edu/~thc/#solutions* (2016).

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms* (3rd. ed.). MIT Press, Boston, MA.

N. G. de Bruijn. 1946. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen* 49 (1946), 758–764.

P. Diaconis and R. Graham. 2012. *Magical Mathematics: The Mathematical Ideas That Animate Great Magic Tricks*. Princeton University Press, Princeton NJ.

L. Euler. 1736. Solutio problematis ad geometriam situs pertinentis. *Comment. Academiae Sci. I. Petropolitanae* 8 (1736), 128–140.

R. P. Feynman. 1965. The Development of the Space-Time View of Quantum Electrodynamics. *Nobel Prize Lecture for Physics* (1965).

R. P. Feynman. 1992. Cargo Cult Science, 1974 CalTech commencement address. In *Surely You're Joking, Mr. Feynman! Adventures of a Curious Character*, R. P. Feynman and R. Leighton (Eds.). Vintage, London, UK.

R. A. Fisher. 1925. *Statistical Methods for Research Workers*. Oliver and Boyd, London. 224–233 pages.

M. Fleury. 1883. Deux problèmes de Géométrie de situation. *Journal de Mathématiques Élémentaires* 2 (1883), 257–261.

I. J. Good. 1946. Normal recurring decimals. *Journal of the London Mathematical Society* s1-21, 3 (1946), 167–169. DOI:10.1112/jlms/s1-21.3.167

R. W. Hall. 25 November 2015. Math for Poets and Drummers. *http://people.sju.edu/~rhall/mathforpoets.pdf* (25 November 2015).

D. R. Hanson. 1987. Printing common words. *Commun. ACM* 30, 7 (1987), 594–598.

C. Hierholzer. 1873. Ueber die Möglichkeit einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Math. Ann.* 6, 1 (1873), 30–32. DOI:10.1007/BF01442866

B. Hopkins and R. J. Wilson. 2004. The Truth about Königsberg. *The College Mathematics Journal* 35, 3 (2004), 198–207.

D. E. Knuth. 1986. *The TEXbook*. Addison-Wesley, Boston, MA.

D. E. Knuth. 1992. *Literate Programming*. Center for the Study of Language and Information Lecture Notes, Vol. 27. Stanford University, Palo Alto, CA. DOI:10.1093/comjnl/27.2.97

D. E.. Knuth. 1998a. *Combinatorial Algorithms*. Vol. 4A. Addison-Wesley, Reading MA.

D. E. Knuth. 1998b. *Seminumerical Algorithms* (3rd ed.). Vol. 2. Addison-Wesley, Boston MA.

R. L. Kruse. 1999. Managing large projects with PreTEX: A preprocessor for TEX. In *TEX Users Group Annual Meeting*. 1070–1074.

L. Lamport. 1994. *LATEX: a Document Preparation System: User's Guide and Reference Manual* (2nd. ed.). Addison Wesley, Boston, MA.

R. Mecklenburg. 2004. *Managing Projects with GNU Make* (3rd. ed.). O'Reilly Media.

M. Petkovšek, H. S. Wilf, and D. Zeilberger. 1996. $A = B$. A K Peters, Wellesley, MA. Foreword by D. E. Knuth.

S. Pinker. 2015. *The Sense of Style: The Thinking Person's Guide to Writing in the 21st Century*. Penguin, London.

K. R. Popper. 2002. *The Logic of Scientific Discovery*. Routledge Classics, London.

Project Jupyter team. 2016. Project Jupyter. *jupyter.org* (2016).

C. F. Sainte-Marie. 1894. Solution to problem number 48. *L'Intermédiaire des Mathématiciens* (1894), 107–110.

R. Sedgewick. 1983. *Algorithms*. Addison-Wesley, Reading MA.

R. Sedgewick and K. Wayne. 2011. *Algorithms* (4th ed.). Addison-Wesley, Boston, MA.

R. Sedgewick, K. Wayne, and N. Liu. 2015. DirectedEulerianCycle.java. See Sedgewick and Wayne [2011].

M. Syed. 2015. *Black Box Thinking*. John Murray, London.

H. Thimbleby. 2003a. The Directed Chinese Postman Problem. *Software — Practice and Experience* 33, 11 (2003), 1081–1096. DOI:10.1002/spe.540

H. Thimbleby. 2003b. Explaining Code for Publication. *Software — Practice & Experience* 33, 10 (2003), 975–1001. DOI:10.1002/spe.537

H. Thimbleby. 2004. Give your computer's IQ a boost. *Times Higher Education Supplement* 9 May (2004). http://www.timeshighereducation.co.uk/story.asp?sectioncode=26&storycode=176549

H. Thimbleby. 2012. Heedless Programming: Ignoring Detectable Error is a Widespread Hazard. *Software — Practice & Experience* 42, 11 (2012), 1393–1407. DOI:10.1002/spe.1141

M. S. Turan. 2011. Evolutionary Construction of de Bruijn Sequences. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence, AI Sec'11*. ACM, 81–86. DOI:10.1145/2046684.2046696

Wikipedia. accessed 2016. *Comparison of documentation generators*. http://en.m.wikipedia.org/wiki/Comparison_of_documentation_generators

S. Wolfram. 1999. *The MATHEMATICA Book* (4th ed.). Cambridge University Press, Cambridge.