How to publish algorithms reproducibly and A new elegant algorithm for the design of sequential experiments

Harold Thimbleby, David Williams
University of Swansea, Wales
Correspondence: harold@thimbleby.net

April 12, 2016

Abstract

This paper presents a very short, elegant and portable algorithm for finding Euler cycles which 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 written in a style to simplify porting of the code to other languages.

To explain and present the algorithm, we use and describe a novel "reversed 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. 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.

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

1 Reliable sequential experiments

In a sequential experiment, such as generation of a 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 that are 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).

Our problem, then, is to write a computer program to generate an efficient sequence of experiments for the general case.

For simple cases, however, the problem can be solved by hand by drawing a graph with one vertex for each of the N types of experiment (i.e., in this case, types of wine on our list), and with N^2 arrows linking each type to each of the others. (This creates a complete graph, usually denoted K_N; the background in graph theory is explained in section 2.1 below.) Each arrow then represents a possible sequence of experiments. Figure 1 shows the graph for $N = 3$.

A sequence of experiments forms a path in this graph following appropriate arrows, and an optimal sequence is a shortest such path. It is a standard graph theory result that for a complete graph that the shortest sequence only needs to follow each arrow exactly once, and will also end up where it started — obviously, if it followed any arrow more than once, it would be repeating an experiment unnecessarily. An optimal sequence is called an *Euler cycle*, after Leonard Euler who first studied paths in graphs (but not sequential experiments). Finding Euler cycles is a well-known programming problem, which has been solved many times. Our problem now reduces to finding an Euler cycle algorithm.

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 the experiment, rather than learn how to program algorithms! Worse, in many textbooks and published papers about the Euler cycle problem, the algorithm is presented as a sketch or in a simplifying pseudo-language. While this may be sufficient to explain the principles or to estimate the time complexity of the algorithm, or to do other things programmers are interested in, it is inadequate to convert into a working program in an executable language such as C or Java. To get a working program, often a lot of extra detail needs carefully working out: how should a graph be represented in a language with type checking, for instance? Or should you use a standard package, and then have to convert the pseudo-program into the programming conventions of the package?

Furthermore, Euler cycle programs assume the graph is arbitrary (and there are different algorithms for directed and undirected graphs). In our case, the graph happens to be both directed and complete, and we discovered that these facts can be used to dramatically 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* program, as will be fully described in this paper (shown in full section 3.2), is just 16 lines of code (plus 5 lines of test code); moreover, deleting a purely diagnostic routine would save another 3 or 4 lines, leaving just 12 "real" lines. This compares very favourably to the reference algorithm [25] that has 22 lines of code (plus 49 lines of test code); and even then this reference algorithm will not work "out of the box" without also finding and compiling it with the various Java and data files it depends on — the total code needed runs to 493 lines.[1] This reference code also requires minor work to edit the Java and local Java environment to compile it, as well as to define the 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 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 scientists who are not familiar with Java

The Java reference algorithm is sophisticated, but there is an evident separation of the published book text [24] from the published program [25], which allowed the program to develop into a sophisticated program at the unintentioned cost of losing clarity and reproducibility in the published book. In fact, the book [25] omits the code altogether as it is too long and complicated: it has to be downloaded from the web instead. Of course, this is entirely defensible: the web code has many details (e.g., features for teaching) that go far beyond

[1] All line counts ignore comments and blank lines.

what is needed in a book — and the code is complete with unit tests and examples, which would be long and tedious to explain in a book, and even a source of error when readers try to copy something so long by hand. However, it is clear that normal publishing practices do not help the goal of publishing easily reproducibe algorithms.

We describe our new algorithm and its applications next, in section 2, and subsequently describe our new approach to publishing algorithms, in section 3.

2 A new Euler cycle algorithm

Many apparently simple algorithms to generate Eulerian cycles have been described, such as Hierholzer's algorithm [12] (invented in 1873, well before modern computers) which finds cycles in the graph and merges them together one by one. However their simple descriptions in English belie their comparatively complex implementations in software — such as just saying "depth first search" to find cycles and doubly-linked lists to "merge" them. What are the details of depth first search or merging? This common but deceptive simplicity is a problem we also noted in our previous discussion of the Chinese Postman Tour [27].

We therefore propose a novel simple and portable algorithm to generate Eulerian cycles on directed complete graphs. It is intended to be used to conveniently generate de Bruijn sequences for experimental study design, as we will explain.

2.1 Graph theory background and applications

Graph theory can be applied to our problem of experimental design to generate a sequence of experiments which is efficient and normalises the effects of random and systematic error.

Leonard Euler effectively invented graph theory in 1736 with his solution to the famous Königsberg[2] bridge problem [5, 13]: 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 every pair of distinct vertices is connected by an edge in each direction. (A complete graph of 3 vertices is shown in figure 1.)

[2]Königsberg was in Prussia and is now Kaliningrad, Russia.

A directed graph is *strongly connected* if it contains a directed path $u \rightsquigarrow v$ (i.e., from vertex u to vertex 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.

> *Notation*. We use $u \to v$ for the directed edge directly connecting vertex u to vertex v, and $u \rightsquigarrow v$ for a directed path, consisting of one or more edges, connecting u to v. Vertices connected by one edge are said to be *adjacent*. A *cycle* is a path that starts and ends at the same vertex.

Graphs represent relations. A complete graph relates every vertex to every other vertex, and therefore represents an *equivalence* relation. When every vertex in a graph is connected to itself via an edge, this represents a *reflexive* relation; and when all pairs of adjacent vertices $u \to v$ are linked by reciprocally directed edges $v \to u$, this represents a *symmetric* relation. If for all paths $u \rightsquigarrow v$ there is an edge $u \to v$, then the graph represents a *transitive* relation.

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 [3] in 1946, they had been previously described in the 19th century [8, 22].

It is possible that the Sanskrit poet Pingala employed a $(2, 3)$ de Bruijn sequence over 1,000 years ago to permute poetic meters and drum rhythms [10, 16]. 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 [4].

An Eulerian cycle may be used to generate de Bruijn sequences [8, 12]. An Eulerian cycle of a complete graph with N vertices will follow every edge $u \to v$ 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 tour is an Euler cycle generating the corresponding de Bruijn sequence; as a special case, the complete graph is a de Bruijn graph — reference [9] gives some examples. Other approaches for generating de Bruijn sequences include feedback shift registers and genetic algorithms [16, 31].

Fisher proposed the application of randomised Latin Squares to balance and equalise sampling error and bias in the design of experiments into soil fertility [7]. 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 even in improving the signal:noise ratios of MRI (Magnetic Resonance Imaging) scans [1].

2.2 The new algorithm

We make five observations:

1. We can use a recursive algorithm to generate an Euler cycle which does not rely on an explicit data structure 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.

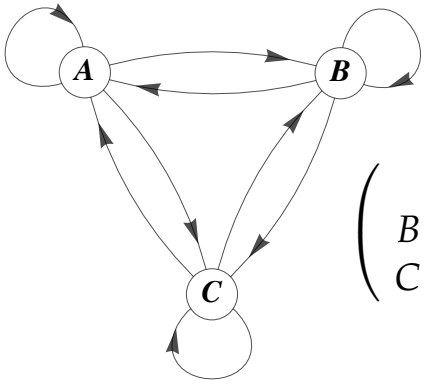
2. Instead of using a complex algorithm to generate cycles, we can consider all trivial cycles of length 1 and 2. For a complete graph ($N > 2$), for any two different vertices u and v, there is one cycle of length 2: namely $u \to v \to u$; and two cycles of length 1: namely $u \to u$ and $v \to v$.

3. There are only two types of trivial cycle. Hence the recursive algorithm for following a cycle only needs to distinguish these two cases, which is trivial since it depends only on whether $u = v$.

4. Once a cycle has been walked, it should not be walked again. The algorithm therefore starts by initialising every trivial cycle as unwalked, and as the recursive algorithm walks a cycle it marks it as walked, and hence will not walk it again.

5. Walking trivial cycles can be easily represented by a two dimensional Boolean matrix as follows: $walked_{uv}$ is true if the cycle $u \to v \to u$ has been walked (and if $u = v$ then $walked_{uu}$ is true and the cycle $u \to u$ has been walked). We call this representation of a graph a *cycle matrix*.

   Note that *walked* does not represent the edges of the graph; rather it represents the graph's trivial cycles, and whether they have been walked.



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

Figure 1: A representation of the complete directed graph K_3 (shown left) for 3 vertices, A, B, C, where there is an edge $u \to v$ for all pairs of $u, v \in \{A, B, C\}$. The graph is composed from 6 trivial cycles represented explicitly in the cycle matrix (shown right). In the diagram, three trivial cycles are single-arrow loops (e.g., $A \to A$), and three trivial cycles are double-arrow loops (e.g., $A \to B \to A$). The matrix has one entry for each trivial cycle; the three missing entries in the matrix are not required, as, for example, the top right would be $A \to C \to A$ but this is exactly the same cycle as $C \to A \to C$, which is shown bottom left.

> Since any trivial cycle $u \to v \to u$ is the same as $v \to u \to v$, the cycle matrix is symmetric, and could therefore be represented as a triangular matrix (as shown in figure 1). However we treat *walked* as a symmetric matrix (i.e., $walked_{uv} = walked_{vu}$) as this simplifies the implementation — but using more memory.

Hence the following algorithm (written in C) is suggested. The Euler walk is a simple recursive algorithm:

```
void walk(int u, int v) // follow a cycle starting u -> v
{   recordEdge(u, v); // record the edge walked
    walked[u][v] = walked[v][u] = 1; // keeps the matrix symmetric
    // find all unwalked cycles starting at vertex v
    for( int w = 0; w < N; w++ )
    {   if( !walked[v][w] )
        {   walk(v, w); // start of either a cycle of length 1 or 2
            if( w != v ) // if a cycle of length 2
                walk(w, v); // walk the rest of it
```

```
        }
    }
}
```

In this code, vertices are numbered 0 to $N-1$. `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)
{   printf("(%d, %d)\n", u, v); // print an edge walked
}
```

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

```
walk(0, 0); // walk starting from 0 returning to 0
```

Calling `walk(0, 0)` assumes the sequence always starts and ends at 0. We discuss alternatives in the next section, below, to randomise the cycle and to avoid always starting a cycle on the same edge.

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

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

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

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

2.3 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. Therefore, for experimental design, we need to modify our algorithm so that the search for an unwalked cycle to recursively follow is randomised. Karl Popper called such sequences "shortest random-like sequences" [21]. The easiest way to do this is to use a random permutation in `walk`'s `for`-loop, as follows:

```
void walk(int u, int v) // follow a cycle starting u->v...
{   recordEdge(u, v); // record the edge walked
    walked[u][v] = walked[v][u] = 1; // keeps the matrix symmetric
    // find all unwalked cycles starting at vertex v
    for( int w = 0; w < N; w++ )
    {   int randomisedw = permutation[v][w];
        if( !walked[v][randomisedw] )
        {   walk(v, randomisedw);
            if( randomisedw != v )
                walk(randomisedw, v);
        }
    }
}
```

Here, `permutation[v]` provides a randomised permutation of the numbers 0 to $N-1$, the value of `w` being mapped by `permutation[v][w]` to a random value in that range. There are N independent permutations (indexed by `v`) to avoid any dependencies between choices made at each vertex. Unfortunately, this simple randomisation is not completely uniform, since only the outgoing edge from v is randomised; the return to v always happens last at the randomised vertex. Hence if $u \to v, u \neq v$ is the first entrance to v, then $v \to u$ will be the last exit from v. For most applications, this is at most a merely technical problem; in particular, because an Euler cycle is a cycle, it can be rotated and started at any point, and this trick can be used to change the "last" exits for any vertices.

The Knuth-Fisher-Yates shuffle [17, 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 = randIntTo(v+1);
            permutation[u][v] = permutation[u][randomv];
            permutation[u][randomv] = v;
        }
    }
```

To avoid always starting an Euler cycle at the same vertex, the base call `walk(0, 0)` (which always starts and finishes at 0) must be randomised too:

```
walk(randomU = randIntTo(N), randomV = randIntTo(N));
if( randomU != randomV )
    walk(randomV, randomU); // get back to randomU
```

Since the Knuth-Fisher-Yates shuffle is carefully designed to generate a uniform distribution of permutations, the choices made in the modified `walk` will be uniformly random. Therefore the series of randomised walks generated by the above code will be uniform samples of all possible Eulerian cycles.

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

This code assumes $n \ll$ `RAND_MAX` $\approx 2^{31}$ [17, p119]. Note that 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));
```

2.4 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:[3]

```
// 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 the vertices rather than their numbers:

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

[3]Not shown here, but our wine sequence generating program, which is generated automatically from this paper using reversed literate programming, 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.)

... 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 used LaTeX's `$\rightarrow$` to generate a "→" symbol to make the experiment sequence look a bit neater when typeset by LaTeX. With these changes, the code generates the following sequence: N = 3 is correct for the number of wines declared. Merlot → Merlot → Pinotage → Pinotage → Shiraz → Shiraz → Merlot → Shiraz → Pinotage → Merlot.

It is easy to confirm by hand that this is indeed an Euler cycle (perhaps by ticking off the edges in figure 1). Note that, as an Eulerian cycle ends at the starting vertex, an additional drink of (in this case) The wine experiment sequence was generated automatically from 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 (twice) for this paragraph by using Unix's `tail -n 1 winelist.tex`. is required — thus a balanced scientific experiment would repeat experiments with randomly selected starting vertices each time (using the ideas of section 2.3). In our example, this would avoid the potential bias caused by drinking too much Merlot — in the sequence above Merlot happens to be drunk both first and last, and hence once more than any other wine.

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 take it for granted that algorithms are described in English, in psuedocode, 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 [29, 30].) 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 [6].

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 [21].

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.

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. Then, because we were excited by our algorithm, we started to write this manuscript. We used the typesetting system LaTeX 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.[4] 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.

This idea is very similar to *literate programming* (Knuth [15]) 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 [28]. Our earlier paper in *Software—Practice and Experience* on the related Chinese Postman Tour [27] used **warp** to present accurate Java code.[5] **Loom** [11] (originally written by Janet Incerpi and Robert Sedgewick for their classic algorithms book [23]) is another approach, similar to **warp**, but allows the use of Unix filters to per-

[4] Unfortunately wines starting with u, v, w, are not commonly recognisable.

[5] The Chinese Postman finds the shortest cycle in a graph that is not necessarily Eulerian; it is a non-trivial generalisation of the problem discussed in this paper.

form arbitrary transformations of code (e.g., to handle special symbols) that is then inserted into arbitrary documents.

Mathematica, which we started out using, is a special case of literate programming. In Knuth's form, the entire program is presented, but in an order that suits exposition. In Mathematica, the entire program (or most of it) is presented, but it has to be in the right order Mathematica requires (e.g., declarations precede uses, even if the need for a variable definition makes little sense until it is used). Mathematica constrains how an author can write, and Knuth's literate programming demands an author write about everything. In general, neither is ideal. In the end, Mathematica is often used for presenting results (hiding the program details) and literate programming is mostly used for internal documentation (disclosing all details).

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 fully automatic process that goes from this paper to executable code. What you now see may not be all of the code,[6] but the code shown *does work* exactly as shown. (In fact, there is nothing in the approach that ties the output to an executable file; the same method can be used to automatically generate an output file in any format from a LaTeX document source file.)

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 reversed literate programming to the highly sophisticated, such as PreTeX [18] that are designed for large complex projects and have commensurate learning curves; for a review see [28].

3.1 Reverse literate programming — the details

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

```
\begin{verbatim}
printf("Hello world!\n");
\end{verbatim}
```

[6] There is exactly one line missing; see section 3.2.

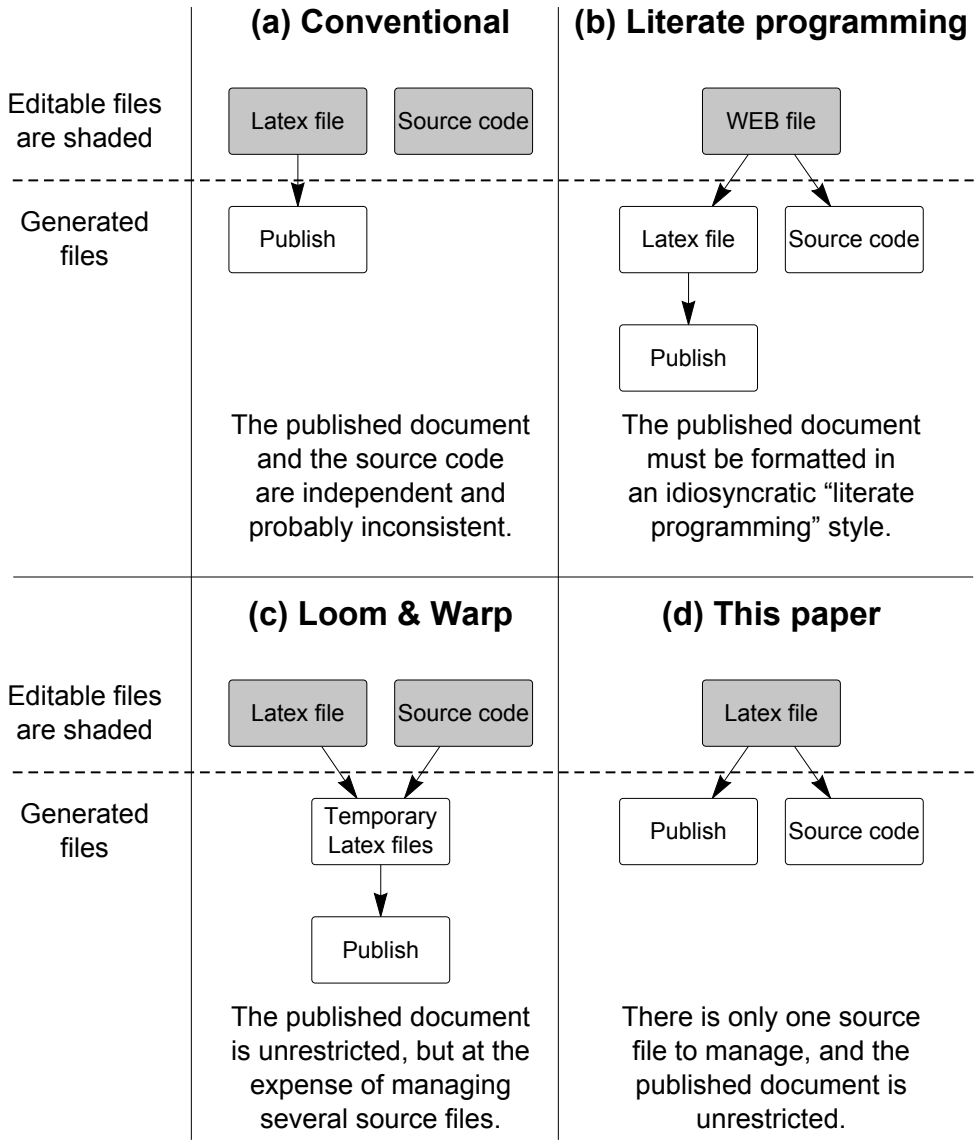


Figure 2: Comparing various ways to write about programming. Note that in the conventional approach (a) there is no guarantee that the published paper faithfully represents the source code, as the paper and source code can be (and will be) edited independently: what is published has no automatic connection 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.)

There is nothing unusual in this. However, we need to be able to generate a compilable program from such code snippets, even though they may be scattered 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 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}
printf("Hello reversed literate programming!\n");
\end{verbatim}
```

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

The document can generate any number of source code files, by writing 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 `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 reversed literate programming enables us to write a paper *and* check whether the code we are writing is valid.

As a matter of fact, we added the following code to the paper only after we had implemented reverse literate programming. Reverse literate programming helped us find an error in our description: both declarations and a bit of wider context are 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:

```
%generate euler.c ., /end/-1
#define N 5 // for a graph with N vertices
<common-declarations>
<declare-walked>
<define-basic-recordEdge>
<define-non-randomised-walk>

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

It is unlikely that a paper would present the above code explicitly: in this paper, this `%generate` command could have appeared after LaTeX's `\end{document}` (so it would disappear and not be typeset as part of the paper), as can any "hidden" definitions for any details that need not be part of a typical published paper. There is no restriction on the order of generating files: they can be specified in any order that best suits the needs of the author. However, because the code was written above, as shown, it actually works *exactly* as written: it generates the file `euler.c` which was used to check the code presented in this paper.

Name definitions in **relit** can occur before or after they are used; in this paper the definition of the common declarations (standard header files, etc) appears after the end of the LaTeX document and is therefore not typeset anywhere in this paper: 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 if the paper is to be able to automatically generate executable code. An alternative approach that could be used with C is to only generate "interesting" part of the code and use C's `#include` from another C file to include the generated code made from

LaTeX into a bigger program that provides the correct context to compile and run them.

In general, authors of papers may want arbitrary text generated from programs (for example, we showed the text generated by running our algorithm on a selection of wines), and the generated text may require sophisticated testing. 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 2.4), using `\input` to read in the program's saved output to insert it into the paragraph where it was needed.

The approach is both very general and ensures *everything* is kept in one place in a single LaTeX source file, where it is easy to edit reliably.

Names can also be defined in the tool'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. Of course, files created by reversed literate 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 `\&`).

Our reverse literate programming tool **relit** provides all the normal checks, such as reporting if names are defined and never used, used recursively, etc. Indeed, some authors have used deliberate "errors" as a way of providing meta-comments: 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 the author to fix a problem with the document.

Relit is a short C program, 270 lines long.

3.2 Checking hidden code

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" [20]: 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 issues.

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 has been hidden. 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, where the relevant tags are output as each *name* (or file) 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 <*name*> then they will be 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 several lines of text.

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

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.	No diagnostics possible.
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. Better for documentation than peer reviewed publications.	No diagnostics needed. Fewer files 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.	No diagnostics.
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.	Diagnostics. Fewer files to maintain.

Figure 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 3.2) — and what is not published remains in the original LaTeX files, so the authors always remain aware of it.

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

```
%set-tag tag
```

That *tag* is then applied to all subsequent definitions (and files) until it is superceded, or overridden by explicit tags. Hence, typically a LaTeX document will start:[7]

```
...
\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.[8]

[7] In fact, `%set-tag` can be used any time earlier than `\begin{document}` too, but it may get lost in LaTeX's own declarations and other preliminaries.

[8] 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.

The code shown below is in fact one of the example programs discussed in this paper: the highlighted text reveals code that was not shown in the paper (except here of course) but which was generated for the compiled test programs. The authors of this papers 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 5 // for a graph with N vertices
#include <stdio.h>                              *** hidden ***
int walked[N][N]; // to represent a complete graph with N vertices
void recordEdge(int u, int v)
{   printf("(%d, %d)\n", u, v); // print an edge walked
}
void walk(int u, int v) // follow a cycle starting u -> v
{   recordEdge(u, v); // record the edge walked
    walked[u][v] = walked[v][u] = 1; // keeps the matrix symmetric
    // find all unwalked cycles starting at vertex v
    for( int w = 0; w < N; w++ )
    {   if( !walked[v][w] )
        {   walk(v, w); // start of either a cycle of length 1 or 2
            if( w != v ) // if a cycle of length 2
                walk(w, v); // walk the rest of it
        }
    }
}
int main(int argc, const char *argv[])
{ walk(0, 0); // walk starting from 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 … bar the one line, shown above in the diagnostic output from **relit**, that we consider 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 the assurances of reproducibility.

3.3 The benefits of automatic honesty

Feynman [6] warned about the utter honesty essential to do good science in 1974, when 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."

But now we can go further: we can help make honesty automatic. This is so important: when honesty is automatic, it is easy. 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. 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 [26].

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 together.

It must be pointed out that not everybody agrees under all circumstances. For example, Knuth writes [14, p*vii*]:

> "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."

His argument is that a reader learns, and at first things need to be simplified with white lies as he calls them [14, 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 advantages. Another point of view is that perhaps Knuth should have improved TEX so it did not require lying to explain it well.[9]

While the *TEXbook* is an exceptional piece of writing, elsewhere Knuth himself has said science is what we can explain well enough to computers [19], and that everything else is art.

Our position is that if people are publishing scientific papers that cannot "be explained to computers" — when they describe algorithms, they surely ought to work as described — then their papers are too obscure to be called science. 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.

4 Conclusions

We developed an algorithm to help perform reliable sequential experiments, and in the process we also developed a tool to help publish reliable papers about algorithms.

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 research.

[9] 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.

Conventional literate programming prevents transcription errors by linking source code and manuscripts. Our novel "reversed 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 description in the paper at will, and repeatedly automatically generate an executable program we could then run and use to confirm the integrity of the code we have published. For example: at the last moment, Paul Cairns pointed out that we had confused wine regions (e.g., Chianti) for grape varieties (e.g., Pinotage and Merlot); we therefore did a *simple* global edit to replace our original dubious use of Chianti with Shiraz. This simple change *in one file*, also, thanks to using **relit**, recreated a new C program that used Shiraz, and then inserted the output from running it back into this paper so the example output (used in section 2.4) was also automatically updated to say Shiraz instead of Chianti.

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

Acknowledgements

This research was funded by EPSRC grant no. [EP/L019272/1]. We are grateful to Paul Cairns for making us greatly improve the paper.

References

[1] G. K. Aguirre, M. G. Mattar, and L. Magis-Weinberg. de Bruijn cycles for neural decoding. *Neuroimage*, 56(3):1293–1300, 2011.

[2] G. Chartrand and L. Lesniak. *Graphs and Digraphs*. Chapman and Hall, 3rd edition, 1996.

[3] N. G. de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49:758–764, 1946.

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

[5] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Comment. Academiae Sci. I. Petropolitanae*, 8:128–140, 1736.

[6] R. P. Feynman. Cargo cult science, 1974 CalTech commencement address. In R. P. Feynman and R. Leighton, editors, *Surely You're Joking, Mr. Feynman! Adventures of a Curious Character*. Vintage, 1992.

[7] R. A. Fisher. *Statistical Methods for Research Workers*. Oliver and Boyd, 1925.

[8] M. Fleury. Deux problèmes de géométrie de situation. *Journal de Mathématiques Élémentaires*, 2:257–261, 1883.

[9] I. J. Good. Normal recurring decimals. *Journal of the London Mathematical Society*, s1-21(3):167–169, 1946.

[10] R. W. Hall. Math for poets and drummers. `http://people.sju.edu/~rhall/mathforpoets.pdf`, 25 November 2015.

[11] D. R. Hanson. Printing common words. *Communications of the ACM*, 30(7):594–598, 1987.

[12] C. Hierholzer. Ueber die möglichkeit einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. *Mathematische Annalen*, 6(1):30–32, 1873.

[13] B. Hopkins and R. J. Wilson. The truth about Königsberg. *The College Mathematics Journal*, 35(3):198–207, 2004.

[14] D. E. Knuth. *The TEXbook*. Addison-Wesley, 1986.

[15] D. E. Knuth. *Literate Programming*, volume 27 of *Center for the Study of Language and Information Lecture Notes*. Stanford University, 1992.

[16] D. E.. Knuth. *Combinatorial Algorithms*, volume 4A. Addison-Wesley, 1998.

[17] D. E. Knuth. *Seminumerical Algorithms*, volume 2. Addison-Wesley, 3rd edition, 1998.

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

[19] M. Petkovšek, H. S. Wulf, and D. Zeilberger. $A = B$. A K Peters, 1996. Foreword by D. E. Knuth.

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

[21] K. R. Popper. *The Logic of Scientific Discovery*. Routledge Classics, 2002.

[22] C. F. Sainte-Marie. Solution to problem number 48. *L'Intermédiaire des Mathématiciens*, pages 107–110, 1894.

[23] R. Sedgewick. *Algorithms*. Addison-Wesley, 1983.

[24] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011.

[25] R. Sedgewick, K. Wayne, and N. Liu. Directedeuleriancycle.java. In `http://algs4.cs.princeton.edu/42digraph/DirectedEulerianCycle.java` [24].

[26] M. Syed. *Black Box Thinking*. John Murray, 2015.

[27] H. Thimbleby. The directed chinese postman problem. *Software — Practice and Experience*, 33(11):1081–1096, 2003.

[28] H. Thimbleby. Explaining code for publication. *Software — Practice & Experience*, 33(10):975–1001, 2003.

[29] H. Thimbleby. Heedless programming: Ignoring detectable error is a widespread hazard. *Software — Practice & Experience*, 42(11):1393–1407, 2012.

[30] H. Thimbleby. Give your computer's IQ a boost. *Times Higher Education Supplement*, 9 May, 2004.

[31] M. S. Turan. Evolutionary construction of de Bruijn sequences. In *AI Sec'11. Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, pages 81–86, 2012.

A Brief correctness argument

If a directed graph G is complete, then for all vertices $u, v \in G$:

$u \neq v$: there is an edge $u \to v$ for every pair of u, v and there is also an edge $v \to u$. The pair of edges $u \to v$ and $v \to u$ form a trivial cycle, namely $u \to v \to u$.

$u = v$: there is an edge $u \to u$ which forms a trivial cycle.

The initialised cycle matrix therefore correctly represents a complete graph as a composition of trivial cycles. Moreover the matrix is symmetric.

There are many equivalent characterisations of graphs with Eulerian cycles. We use the following: a directed graph has an Eulerian cycle if and only if: (i)

it is strongly connected, and (*ii*) it can be decomposed into edge-disjoint (non-overlapping) directed cycles (i.e., each edge is part of exactly one cycle). A proof uses Veblen's Theorem [2].

As `walk` recursively expands trivial cycles, it effectively composes the initial decomposition into trivial cycles into an Euler cycle. Therefore the algorithm will print a cycle. Because of the `walked` test, no edge is walked more than once. The next part of the proof shows that when the walk terminates there are no missed cycles.

After walking an edge, $u \to v$, in each iteration of the `for` loop, the body of walk recursively explores every connected vertex from v. If we assume no cycles have been walked, then because the graph is complete, every other vertex w will be visited and hence all cycles starting with each vertex will be explored. Hence, the very first call of `walk` is sufficient to walk every cycle.

Interestingly, provided the graph remains strongly connected, deleting trivial cycles from the cycle matrix makes no difference to the argument for correct termination. In particular, the algorithm will work correctly regardless of self-cycles $u \to u$; the graph need not be reflexive, and thus it can find Euler cycles of irreflexive complete graphs.