Constructive complexity: a metric for operations research spreadsheet model design

John F. Raffensperger*

Department of Management,
Private Bag 4800,
University of Canterbury,
Christchurch 8140, New Zealand
E-mail: john.raffensperger@canterbury.ac.nz
*Corresponding author

Pascal Richard

Laboratory of Applied Computer Science,
LISI/ENSMA and University of Poitiers,
Téléport 2 – BP 40109,
Futuroscope 86961, France
E-mail: pascal.richard@univ-poitiers.fr

Abstract: This article makes a modest step toward a quantitative measure of the work required to solve a specific kind of problem: a given spreadsheet design. The method relies on a measure that we define called constructive complexity, which is the number of keystrokes required to write a spreadsheet for a given computational task, as a function of the input data. The notion of constructive complexity is simple but general. The measure is simple enough to be described briefly in the classroom and the textbook. As an example, we use the shortest path problem. The objective is to educate operations research students and practitioners to write better spreadsheets, and to evaluate the quality of different spreadsheet designs. Constructive complexity could be used as a tool to achieve these objectives. We do not claim that it resolves all problems of spreadsheet design, but it is one step forward in quantifying spreadsheet design, and is intended to generate additional debate.

Keywords: computational analysis; decision support systems.

Reference to this paper should be made as follows: Raffensperger, J.F. and Richard, P. (2008) 'Constructive complexity: a metric for operations research spreadsheet model design', *Int. J. Information and Operations Management Education*, Vol. 2, No. 4, pp.388–406.

Biographical notes: John Raffensperger is a Senior Lecturer in management science at the Department of Management, Christchurch, New Zealand. His research includes problems of environmental sustainability, smart markets for water resources and combinatorial optimisation.

Pascal Richard is a Professor in computer science at the Department of Business and Public Management, Institute of Technology, University of Poitiers, France. Teaching interests include spreadsheet modelling for operations research problems and database systems. He also works with the

Copyright © 2008 Inderscience Enterprises Ltd.

Laboratory of Applied Computer Science at the National Engineering School of Mecanics and Aerothechnics, Poitiers. His research interests include real-time systems, scheduling theory, on-line algorithms and combinatorial optimisation.

1 Introduction

Problem solving is often done in spreadsheets, so spreadsheet writing is a growth industry. The latest operations research textbooks make heavy use of Excel, Solver, What'sBest, CrystalBall, @Risk, management science (MS) Project and the like. Operations research teachers and practitioners need to stay on top of the latest techniques in spreadsheets in order to provide the best tools to their students and clients. However, despite the variety of work and strong opinion, writing and auditing spreadsheets remain subjective and error-prone tasks.

Researchers are already familiar with many horror stories about errors in spreadsheets, which we shall not repeat here. Guidelines for spreadsheet design have been suggested by a long list of researchers, including Bromley (1985), Kee (1988), Kee and Mason (1988), Stone and Black (1989), Bissell (1986), Edge and Wilson (1990), Crain and Fleenor (1989), Cragg and King (1993) and Ragsdale (2004; p.63). The last three are quite different from the others, with less focus on checklists of features to add and more focus on readability; time will tell whether they bear fruit. But, why has none of these methods taken hold?

We could look to the user as the barrier. Cragg and King (1993) found that the typical spreadsheet user does not have a background in data processing. Furthermore, both Cragg and King (1993) and Davis (1996) found that managers use spreadsheets in order to have freedom from their firms' information technology groups, and managers tend to be hostile to perceived interference. Attribution theory kicks in, and the spreadsheet writer fervently believes his or her own spreadsheet is fine, but their neighbour's spreadsheet is not. In short, people do not want to be told how to design their spreadsheets. The solution, of course, would be better education in spreadsheet design, but better education requires good guidelines in the first place, to teach operations research students and practitioners to write better spreadsheets. Even the phrase 'spreadsheet design' may be ambiguous, as it could refer to formatting, layout or even choice of decision model. Here, we will focus on choice of formulas and some aspects of layout, for a given decision model.

We could also look to the guidelines as the barrier. But guidelines rely primarily on the researchers' opinion as to what makes a good spreadsheet. Researchers (and practitioners) have relied on analogies to other fields, most often computer programming (e.g. the first seven references above), but also writing, mathematical notation and graphics (Cragg and King, 1993), to motivate a particular style. Ward (1989) makes an excellent case for simple model construction in general (not just in spreadsheets). His ideas apply beautifully to spreadsheets, even using similar language, e.g. 'constructively simple'. Nevertheless, to our knowledge, no research exists that shows scientifically whether one spreadsheet design is 'better' than another.

This article makes a modest step toward a quantitative measure of the quality of a given spreadsheet design. We have by no means come to the end of the problem, nor do we claim that our method always reduces cognitive complexity, but it is one step forward

in quantifying spreadsheet design. The measure is simple enough to be described briefly in the classroom and the textbook and applies to any kind of spreadsheet. We will try to be honest about the limitations of constructive complexity.

In operations research and computer science, the classic measure of the quality of an algorithm is the algorithm's computational complexity. We introduce an analogous concept of constructive complexity as the amount of work to implement a given design of a spreadsheet. Roughly speaking, constructive complexity refers to the number of unique formulas, the complexity of each formula, and the ease of revising a spreadsheet, as a function of the input data. We shall see that constructive complexity is not only the number of unique formulas. Constructive complexity may help the spreadsheet writer think about layout for a given model, rather than simply cranking out a spreadsheet that seems to work. We shall give a more precise definition later.

The time spent to design a spreadsheet model is almost always much greater than the running time. End users are not computer scientists and developing spreadsheet models is never a prime interest (Conway and Ragsdale, 1997). End users want to design spreadsheet models as simply as possible without errors. But even in general modelling, novices are not good at self-regulation (Powell and Wellemain 2006). Some issues in spreadsheet design include the following.

- Minimise the amount of time to create the spreadsheet correctly. As a proxy for this, a user may wish to minimise the number of keystrokes required to create the spreadsheet correctly. We focus on the number of keystrokes, because we can measure this independently of the user. It may be that additional time, but with fewer keystrokes (not just fewer distinct formulas), will lower the error rate.
- Be able to revise the model easily to solve another instance size. The role of 'what if' is a key point of success of the spreadsheet paradigm.
- Be able to document and audit a spreadsheet easily.
- Enter 'simple' formulas in every cell. We are less concerned with formula length, because a long formula may be quite simple. Simple formulas are presumably easier to understand in isolation, and errors are more likely in complicated formulas. Yet often a huge spreadsheet of hundreds of simple formulas can be reduced to a much smaller spreadsheet that still has reasonably simple formulas. The trade-off may be viewed as simplicity in each cell versus unwieldiness of the whole spreadsheet. In any case, 'simplicity' appears to be subjective, and dependent on the user's knowledge of spreadsheets and familiarity with the particular problem the spreadsheet is trying to solve.

Our concern is to improve the construction of spreadsheets, make them easier to write and audit and lower the number of errors. We shall see that constructive complexity answers these issues only partially, and much more remains to be done. Our contribution is mainly in the first issue, specifically quantifying the minimum number of keystrokes required to write a spreadsheet. The objective is to educate operations research students and practitioners to write better spreadsheets, and to evaluate the quality of different spreadsheet designs.

The article is organised as follows. Section 2 reviews computational complexity. Section 3 defines constructive complexity. Section 4 presents the well-known shortest path problem (SPP) in five spreadsheet implementations. Finally, Section 5 discusses possible extensions and future work.

2 Computational complexity reviewed

In the academic world, MS/operations research problems are usually centred on algorithms. A given input of the algorithm is called an instance of the problem. The running time of an algorithm is measured through the $O(\ )$ notation. This metric focuses on the worst-case number of operations to perform while considering any size of instances. Such an evaluation is independent from hardware and software. In the $O(\ )$ notation, the running time of an algorithm is defined by a function in the size of the input. Thus, that function indicates how the running time of the algorithm grows as the input size increases. Such a notation only leads to a running time bound. More formally, the computational complexity of an algorithm is $O(f(n))$ if there exist constants N and K, such that for every input size $n \geq N$, the number of executed operations is not more than $K \cdot f(n)$.

For instance, consider the SPP. Let n be the number of vertices (V) and m be the number of arcs (E) in the graph $G = (V,E)$ and a weight function $w : E \rightarrow R$. We want to compute the shortest path from one vertex v to every other vertex in G (we assume there is no negative cycle in G). The Floyd–Warshall algorithm for the SPP is an $O(n^3)$ algorithm. From the design point of view, the Floyd–Warshall algorithm is based on simple operations on matrices. The Ford–Bellman algorithm can be implemented as an $O(mn)$ algorithm. Finally, Dijkstra's algorithm can be implemented with a time complexity of $O(n^2)$, if a list is used to store the current lengths from v to every node during the execution of the algorithm. The complexity can be reduced to $O(m \cdot \log n)$ using a classical heap and to $O(m + n \cdot \log n)$ with a Fibonacci heap (which is a heap made of a forest of trees, where inserting a value and finding minimum is done in $O(1)$). In the particular case of a directed acyclic graph (DAG), the problem can be solved in time $O(n + m)$.

The Floyd–Warshall algorithm can be easily programmed with six source lines in FORTRAN. According to the computational complexity metric, this algorithm is clearly inefficient in comparison with the two others. But, the extra effort to implement a more complex algorithm may not be worthwhile if only one instance of the SPP has to be solved. Instead of concern with computation time, the user is more interested in the time required to program the computer. For most spreadsheets, running time is not a prime concern. As a consequence, the classical computational complexity theory is not useful.

3 Constructive complexity

3.1 Definition of constructive complexity using Big O notation

Studies of spreadsheet error focus on the number of cells in the spreadsheet model. Panko and Sprague (1998), for example, compare error rates in spreadsheets to error rates in lines of program code. The analogy is appealing, especially since spreadsheet writing is so often compared with writing computer programs. However, a computer program is written a character at a time, whereas a large spreadsheet is usually written by copying single formulas to many cells. The spreadsheet error rate is therefore likely related to the number of keystrokes required to design a spreadsheet. More keystrokes will probably result in more errors.

A copy of a given formula will be

1. correct in all destination cells
2. wrong in all destination cells (in which case it is likely to be caught relatively quickly)
3. wrong for some key cells, such as end points.

Thus, the error rate is at least partially related to the manner in which the spreadsheet is constructed, rather than the number of cells *per se*. Therefore, we focus on the number of interactions required to create the spreadsheet, as measured by keystrokes. From a human–computer interface point of view, the idea of the number of keystrokes to do a task is well understood (Card, Moran and Newell, 1983), and our definition of constructive complexity may be viewed as an extension.

We define constructive complexity of a spreadsheet to be the number of keystrokes required to implement a given computational procedure in a spreadsheet, as a function of the input data (i.e. the instance). The notation for constructive complexity is the same as for computational complexity, the 'Big O' notation, to capture the asymptotic behaviour of a function. More formally, suppose $f(x)$ and $g(x)$ are two functions. Then $f(x) = O(g(x))$ if and only if there exist constants N and C such that $|f(x)| \leq C\ |g(x)|$ for all $x > N$. For the constructive complexity of a spreadsheet, x is the number of items data items needed to define an instance, just as it is in computational complexity. However, $f(x)$ is the number of keystrokes required of the spreadsheet writer to create the spreadsheet.

Data entry may depend on the spreadsheet design, but we have always found a way to enter data in the same constructive complexity regardless of the layout. This issue may need revisiting, but we will ignore data entry, treating its cost as sunk.

To prove constructive complexity, a necessary condition is to specify the keystrokes to accomplish the given task. Therefore, constructive complexity for a given task depends on the user interface, the spreadsheet program's features, and the spreadsheet writer's ingenuity. We will assume that a rectangular selection (not necessarily one column or row) is an $O(1)$ operation. Similarly, a rectangular fill is an $O(1)$ operation. This is to avoid being overly pedantic about exact keystrokes. For example, entering values 1, …, 50 takes somewhat less time than entering values 1, …, 30,000, because the fill operation depends somewhat on the number of rows. The exact amount of time further depends on subtle interface features such as mouse acceleration, and the writer's physical coordination in stopping the fill on the exact row. Most spreadsheet writers would not be concerned about the difference. So, we will view entering the first three values, selecting them with a mouse and dragging downwards to the correct last row, as a single keystroke operation, independent of the number of rows. Furthermore, the Excel user can select a block using the GoTo function while holding the shift key, which is truly an $O(1)$ operation.

3.2 The spreadsheet compared to classical programming and math modelling

A program written in a procedural language accepts problem instances of different sizes, so such programs have constructive complexity of $O(1)$. We could not imagine a reasonable case where a program written in a procedural language had constructive complexity that was more than $O(1)$, within reasonable limits (e.g. the size of a static array in FORTRAN). In contrast to a procedural language, writing a linear program by hand in Lindo or CPlex requires constructive complexity that depends on the data and the

chosen linear program formulation. For example, the travelling salesman problem (TSP) with n cities is commonly formulated with $O(n^2)$ variables (Dantzig, Fulkerson and Johnson, 1954) or $O(n^3)$ variables (Dantzig, 1963; p.545). (The TSP may be formulated with n variables in a spreadsheet, as in Ragsdale 2004; pp.403–408.) In Lindo or CPlex, the $O(n^3)$ formulation would take longer to write than the $O(n^2)$, because every constraint must be typed out by hand. Therefore, constructive complexity motivates modelling languages such as Lingo, AMPL or GAMS. The goal of making a model data-independent is to make the model of constructive complexity $O(1)$.

Writing a model with a spreadsheet is much more like writing a model in Lindo or CPlex than it is like writing a model in Lingo, AMPL or GAMS. To an operations researcher, the 'flat' or complete model is familiar, exactly like a model in Lindo or CPlex. Each numeric cell corresponds to a coefficient, a variable or a constraint. We shall see that a given spreadsheet model can be written in different ways, with different constructive complexities.

Microsoft Excel has nothing native analogous to a modelling language. Despite arrays and Visual Basic (VB), Excel has no user-defined function that can fundamentally reduce the constructive complexity. (A clever spreadsheet developer may be able to prove us wrong, perhaps by using VB to return an array.) Jones, Blackwell and Burnett (2003) point out that spreadsheets lack the most basic element that programmers use to control complexity, which is the ability to re-use abstractions, and users make up for this lack through copy and paste of formulas. Interestingly, Paine (2001) developed a language to generate the entire spreadsheet, just as Lingo or AMPL generate a linear program. His code can also reverse-engineer a spreadsheet, to find underlying structure in it. Paine's approach gives a data-independent way to create a spreadsheet.

3.3 Revisional complexity of a spreadsheet

Besides the initial construction, we are concerned with the work required to revise an existing spreadsheet, as a function of the input data. Most often, a spreadsheet must be revised because of change in the number of inputs rather than the magnitude of the inputs. We define the revisional complexity of a spreadsheet as the number of keystrokes required to revise an existing spreadsheet for a change in the number of inputs, given the current number of inputs. Especially for spreadsheets that are used repeatedly, revisional complexity may be at least as important as the constructive complexity. The revisional complexity may be viewed as the rate of change in the constructive complexity. If a spreadsheet has a constructive complexity of $\mathrm{O}(n)$, then if we want to make a spreadsheet for $n + 1$ inputs, then the revisional complexity is at most $\mathrm{O}(n)$, since the spreadsheet can be rewrite in $\mathrm{O}(n)$ time from scratch. Thus, the revisional complexity is always less than or equal to the constructive complexity.

A program can impose a particular revisional complexity on the user. For example, a PERT/CPM program (e.g. Seal (2001)) might allow the user to increase the number of activities, by pressing an 'Add activity' button. This suggests that the revisional complexity is $O(1)$, but that is not the intended meaning. To add 50 activities would require the user to press the button 50 times, so the revisional complexity is $O(n)$. Of course, the program could be rewritten in $O(1)$ time to allow addition of n activities at a time.

While a VB subroutine can reduce the work required to revise a spreadsheet, we will focus on the grid itself. In an extreme case, the spreadsheet writer can use VB as a

procedural language, with the spreadsheet grid as only the input. Then we have nothing to study, since the model is not in the spreadsheet. Furthermore, writing a high-quality VB routine that will manage all possible inputs is often harder than simply modifying the spreadsheet directly. If we can write a spreadsheet that has low revisional complexity, then we can avoid the problem in the first place.

Thus, we may view the overall constructive complexity of a spreadsheet as two values: its initial constructive complexity, and its revisional complexity.

3.4 The length of a formula

The spreadsheet writer may worry about the length of formulas, especially when others must read or audit the spreadsheet. A cell formula of the form $c_{i,j} = c_{1,j} + c_{2,j} + c_{3,j} + \ldots + c_{i-1,j}$ is of constructive complexity $O(i)$, because each address must be typed or selected one at a time. By contrast, an equivalent formula of the form $c_{i,j} = \text{sum}(c_{1,j} : c_{i-1,j})$ is of constructive complexity $O(1)$.

For more complicated formulas, such as calculating the utilisation factor of a server that serves periodic jobs (having a service time of C_i, and a period $T_{i,1}$, $1 \leq i \leq n$), one can use an array formula to achieve an $O(1)$ constructive complexity. Assume that service times are in cells A1:An and periods are in B1:Bn then the utilisation factor is calculated by the array formula: = SUM(A1:An/B1:Bn), using Ctrl + Shift + Enter to enter the array formula.

The number of formulas may be reduced with tools. Johnson (2007) developed an Excel add-in called Spreadsheet Composer to substitute intermediate calculations. Spreadsheet Composer can produce long formulas. It will not simplify the formula algebraically, though a single long formula is easier to simplify than a chain of small formulas. Nor will it substitute out spurious cells of the form $x = y$.

In general, the constructive complexity of a formula is subsumed by the constructive complexity of the spreadsheet. Furthermore, we believe that a spreadsheet of low constructive complexity will tend to have reasonably short formulas, but counter-examples are easy to create. In any case, the length of a formula is more of a concern for readability and the need to see intermediate calculations, rather than a significant issue on its own for constructive complexity.

3.5 Auditing

If each keystroke has a certain probability for introducing an error, lower constructive complexity should result in fewer spreadsheet errors. In this case, a spreadsheet that is constructed in fewer keystrokes is more likely to be correct than a spreadsheet constructed with many keystrokes. This should be a strong motivation to reduce the constructive complexity of spreadsheets. However, we can go a little further.

Attention to the constructive complexity of a spreadsheet can be useful in auditing the spreadsheet. Consider opening a spreadsheet created by someone else. Upon inspection, we find that the spreadsheet has one formula that has been copied throughout a large table. We observe that the spreadsheet is $O(1)$ constructive complexity, even if we know nothing else about the spreadsheet or its meaning, even if all the labels are removed. However, observing that the spreadsheet has $O(1)$ computational complexity requires checking every formula, which takes much longer than $O(1)$. This gives us insight as to why spreadsheets are so hard to audit. Even if a spreadsheet is easy to construct, it seems

to require at least $O(C)$ operations to audit, where C is the number of cells, far more than the amount of input data.

A few practitioners have identified the idea of 'types of formulas' sometimes called schema, and some auditing tools can categorise schema (Nixon and O'Hara, 2001; Paine, 2001). (In Lingo, AMPL or GAMS, a schema is nothing more than an equation definition.) Such tools can find cells with different formulas compared to the surrounding cells, though the tools have limitations. For example, Excel 2000 and Excel XP have such a tool, but the tool does not highlight a constant erroneously entered into the middle of a table of formulas. Formulas can confuse the auditing tool or the auditing tool can confuse us. A spreadsheet writer can easily create examples where observing constructive complexity is even greater than $O(C)$, by hiding information in any of a long list of ways. In these perverse cases, we may have to check all 256 * 65,536 cells in the spreadsheet and various dialog boxes besides.

Attention to constructive complexity allows the possibility of a different form of auditing. A spreadsheet could be documented to contain a certain constructive complexity. An auditor can then check a single formula of a given type, and then copy that formula to the entire block that is supposed to contain the formula of that type. The copy operation would overwrite formulas that were different within the block. (There remains the issue of perversely hidden information, and whether that can be eliminated in a time of given complexity.) The auditor would then be sure that the spreadsheet would be constructed as it was documented to be, and the time required to check the formulas in this manner would be of the same order of complexity as was documented.

In the five examples that follow, we shall touch on these perceptual and interface issues. We shall also see the usefulness of advanced spreadsheet features such as arrays, transposition, and circular references.

4 Five shortest path problem spreadsheet implementations

The SPP is a basic graph theoretical problem. We want to find the shortest path from a given initial node to a given terminal node in a graph $G = (E,V)$, where E is the set of arcs labelled by distance between the two connected vertices and V is the set of vertices. Our example problem will be to find the shortest path from a to g in Figure 1. This problem can be expressed as a linear program and solved by dynamic programming. Our first implementations use only basic Excel functions without add-ins. We conclude with a spreadsheet implementation is based on Excel's Solver. For each implementation, we measure the three dimensions of the constructive complexity.

Figure 1 Example shortest path network

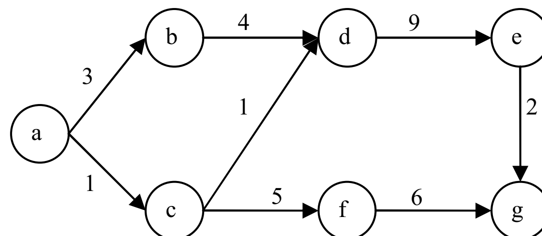


Figure 2 $O(n)$ spreadsheet to solve the shortest path problem

G14 =G6+$D19

	A	B	C	D	E	F	G	H
1	The shortest path problem, O(n) CC.						Formula	
2								
3	From \ To	a	b	c	d	e	f	g
4	a		3	1	99	99	99	99
5	b			99	4	99	99	99
6	c				1	99	5	99
7	d					9	99	99
8	e						99	2
9	f							6
10	g							
11								
12	a	0	3	1	99	99	99	99
13	b			102	7	102	102	102
14	c				2	100	6	100
15	d					11	101	101
16	e						110	13
17	f							12
18	g							
19	Min value	0	3	1	2	11	6	12

Key cell formulas

Cell	*Formula*	*Copied to*	*Calculating*
C12	= C4 + $B19	D12:H12	$c_{a,b} + \min_i c_{i,a}$
D13	= D5 + $C19	E13:H13	$c_{b,c} + \min_i c_{i,b}$
E14	= E6 + $D19	F14:H14	$c_{c,d} + \min_i c_{i,c}$
F15	= F7 + $E19	G15:H15	$c_{d,e} + \min_i c_{i,d}$
G16	= G8 + $F19	H16	$c_{e,f} + \min_i c_{i,e}$
H17	= H9 + $G19	–	$c_{f,g} + \min_i c_{i,f}$
B19	= MIN(B12:B18)	C19:H19	–

4.1 O(n) spreadsheet

A method to solve the SPP can easily be implemented in Excel as in Figure 2. The minimum cost at a node, such as node c, is the minimum total cost into that node from all other preceding nodes. The minimum cost of 1 at node c plus the length of 5 for arc (c, f), results in the total cost of 6 through arc (c, f). Denoting B4 = $c_{1,1}$, then the arc costs are in $c_{1,1}, \ldots, c_{n,n}$. The total arc costs are in $c_{n+2,1}, \ldots, c_{2n+2,n}$. Row $n + 1$ is empty. Therefore, we can write the formula in $c_{n+2,1}, \ldots, c_{2n+2,n}$ as:

$$c_{n+1+i,j} = c_{i,j} + c_{2n+3,i}, \text{ for } i = 1, \ldots, n, \text{ and } j = 1, \ldots, n.$$

In the right hand side, note that i appears in the row of the first term $c_{i,j}$, but in the column of the second argument $c_{2n+3,i}$. The subscript i is therefore transposed from row to column. This transposition is fundamental to many of these shortest path structures,

because the algorithm requires calculation of min($c_{i,j}$ + min$_k$($c_{k,i}$)), which, graphically, is a transposition. The algorithm imposes a graphical effect.

For n nodes, this spreadsheet contains n distinct formulas. Of these, $n - 1$ are in the table B12:H18, with each formula of $O(1)$ constructive complexity. These formulas are distinct by row, because each row in B12:H18 refers to one column of the table B19:H19. One formula in each row must be edited by hand to refer to the correct column in B19:H12 (note the $D19 in the formula bar), and then the formula can be copied to the rest of the row. The nth distinct formula is in the table B19:H19, which is the MIN() over the table B12:H18. Therefore, the whole spreadsheet is of constructive complexity $O(n)$. This spreadsheet is frustrating to write and audit, because of the number of distinct formulas. With so many distinct formulas, the spreadsheet writer is likely to make a mistake. This example suggests that constructive complexity depends mainly on the number of distinct formulas, but this is not always so, as we shall see next.

As an aside, the data requires $O(n^2)$ inputs, but will always require this many inputs for any spreadsheet. Entering this data may be viewed as a sunk cost of making the spreadsheet.

4.2 *O(n) spreadsheet with transpose and array functions*

However, we can simplify the construction of this spreadsheet somewhat. In Figure 3, we used Excel's array (i.e. Ctrl-Shift-Enter) and TRANSPOSE() functions to transpose B19:H19–I12:I18. Now, table B12:H18 has exactly one distinct formula which can be copied to all relevant cells. The whole spreadsheet has three distinct formulas, and can be written more easily.

Figure 3 The shortest path problem spreadsheet has fewer formulas if we transpose the minimum value table, but still requires $O(n)$ copy operations (see online version for colours)

G14 =G6+$I14

	A	B	C	D	E	F	G	H	I	J	K	L
1	The shortest path problem, O(n) CC.							Formula				
2												
3	From \ To	a	b	c	d	e	f	g				
4	a		3	1	99	99	99	99				
5	b			99	4	99	99	99				
6	c				1	99	5	99				
7	d					9	99	99				
8	e						99	2				
9	f							6				
10	g											
11									Min value, transposed			
12	a	0	3	1	99	99	99	99	0			
13	b			102	7	102	102	102	3			
14	c				2	100	6	100	1			
15	d					11	101	101	2			
16	e						110	13	11			
17	f							12	6			
18	g								12			
19	Min value	0	3	1	2	11	6	12				

Key cell formulas

Cell	*Formula*	*Copied to*
C12	= C4 + $I12	D12:H12, E13:H13, F14:H14, G15:H15, H16
I12:I18	{= TRANSPOSE(B19:H19)}	–
B19	= MIN(B12:B18)	C19:H19

Even though we have just three distinct formulas, the upper triangular structure requires $O(n)$ copies. So, the number of distinct formulas is not sufficient to specify the constructive complexity. The revisional complexity of this spreadsheet is $O(n)$, because each new node requires a copy keystroke.

4.3 O(1) version with array functions

By changing the formulas in B12:H18, we can get $O(1)$ constructive complexity. By using an IF() statement in B12:H18, we can avoid a circular reference (unless the adjacency matrix actually contains a cycle). For example, the formula in F15 = IF(F7 < 99,F7 + $I15,99). Note the explicit initialisation of the cost function in cell B12. The constant zero start value in cell B12 can be added after the formula is copied to B12:H18, as in Figure 4.

Figure 4 An $O(1)$ constructive complexity spreadsheet to solve the shortest path problem (see online version for colours)

H21 | =INDEX($A$12:$A$18,MATCH(H19,H12:H18,0))

	A	B	C	D	E	F	G	H	I	J
1	The shortest path, O(1) CC.							Formula		
2										
3	From \ To	a	b	c	d	e	f	g		
4	a	99	3	1	99	99	99	99		
5	b	99	99	99	4	99	99	99		
6	c	99	99	99	1	99	5	99		
7	d	99	99	99	99	9	99	99		
8	e	99	99	99	99	99	99	2		
9	f	99	99	99	99	99	99	6		
10	g	99	99	99	99	99	99	99		
11										
12	a	0	3	1	99	99	99	99	0	
13	b	99	99	99	7	99	99	99	3	
14	c	99	99	99	2	99	6	99	1	
15	d	99	99	99	99	11	99	99	2	
16	e	99	99	99	99	99	99	13	11	
17	f	99	99	99	99	99	99	12	6	
18	g	99	99	99	99	99	99	99	12	
19	Min value	0	3	1	2	11	6	12		
20										
21	Shortest path	a	a	a	c	d	c	f		

Key cell formulas

Cell	*Formula*	*Copied to*
C12	= IF(C4 < 99,C4 + $I12,99)	C12:H18, B13:B18
I12:I18	{= TRANSPOSE(B19:H19)}	–
B19	= MIN(B12:B18)	C19:H19
B21	= INDEX($A$12:$A$18,MATCH(B19,B12:B18,0))	C21:H21

We added the primal retrieval at the bottom of the spreadsheet, a fourth distinct formula. The experienced operations researcher should see that this formula is the primal retrieval for a dynamic program. The ending node is entered in cell I21. A referee observed that the TRANSPOSE() could be replaced by substituting index numbers 1, …, 7 for the node names, a, …, g, and using an INDEX() function. For example, we can write cell C12 = IF(C4 < 99,C4 + INDEX($B$19:$H$19,$A12),99), and copy this to B13:B18 and C12:H18, and thereby eliminate the need for column I. The formulas in row 21 would replace the $A$12:$A$18 references with $A$4:$A$10.

The differences in constructive complexity between the three versions are not clear simply from a printout of the spreadsheet. To show constructive complexity requires describing the keystrokes necessary to construct the spreadsheet. In addition, we see that to construct a spreadsheet in a certain amount of time is not the same as checking it. To check the spreadsheet and to observe that it actually does have a certain constructive complexity, we may have to examine every formula, $O(F)$, or even every cell in the workbook including empty cells. However, given the documentation, an auditor could check one of each distinct formula, and copy it to the other cells as appropriate, thereby being sure that the spreadsheet matches the documentation. This copying can be done in $O(1)$ time, the same as the constructive complexity of the spreadsheet.

To add another node in the spreadsheet, the writer needs to:

1 add a row to the arc length table with the cursor in row 19

2 add a row to the adjacency table with the cursor in row 11

3 add a column to the entire spreadsheet with the cursor in column H

4 copy the formula from G21 to H21 and from G23 to H23

5 copy formulas from H13 to I13:I20 and to B20:I19

6 correct the TRANSPOSE() function (because rows cannot be inserted in an array)

7 finally, correct the data.

The number of steps in adding nodes is independent of the number of nodes, so this spreadsheet has $O(1)$ revisional complexity. Note that the total number of formulas is not the same as the constructive complexity. The first two formulations have $n^2/2 - n$ formulas in cells B12:H18. The last formulation has over twice as many formulas, but a lower constructive complexity. This implementation does not have the fewest cells, since a spreadsheet could be written in only the number of nodes, and another implementation could have lower constructive complexity. Such an implementation might have one distinct formula in each of n cells, with the formula having only $O(1)$ selections, and could therefore be written in fewer than three keystroke operations. We shall see such sparsity soon.

Figure 5 An elegant $O(1)$ spreadsheet for the shortest path problem (see online version for colours)

H7 | {=INDEX($A$4:$A$11,MATCH(F7&G7,$B$4:$B$11&$D$4:$D$11,0))}

	A	B	C	D	E	F	G	H
1	Finding the shortest path in a directed graph.							Formula
2								
3	From	To	Distance	Total length		Node	Path length	From
4	a	b	3	3		a	0	
5	a	c	1	1		b	3	a
6	c	d	1	2		c	1	a
7	b	d	4	7		d	2	c
8	c	f	5	6		e	11	d
9	d	e	9	11		f	6	c
10	e	g	2	13		g	12	f
11	f	g	6	12				

Key cell formulas for Figure 5

Cell	*Formula*	*Copied to*
D4	= VLOOKUP(A4,$F$4:$G$10,2) + C4	D5:D11
G4	{= MIN(IF(F4 = $B$4:$B$11,$D$4:$D$11))}	G5:G10
H5	{= INDEX($A$4:$A$11,MATCH(F5 and G5,$B$4:$B$11 and $D$4:$D$11,0))}	H5:H10

4.4 Arc-oriented version

In our work on dynamic programs, we found we could improve the constructive and revisional complexity of a spreadsheet by using arrays, matrix transposition, circular references and database lookups. We are aware, however, that many end users are not familiar with these advanced functions. However, an advanced function can be used with great elegance. Figure 5 presents a version communicated by an anonymous referee. This spreadsheet has clever use of the MATCH() function, with concatenation in an array formula, $B$4:$B$11 and $D$4:$D$11. This formula concatenates the ‘to’ node with the ‘total length’, and MATCH() function searches over this array to match the ‘node’ and ‘path length’. The spreadsheet is $O(1)$ in constructive and revisional complexity.

Advanced functions played a useful role in the SPP, reducing the constructive complexity from $O(n)$ to $O(1)$. Some users may consider these functions to be advanced spreadsheet programming, and no doubt these are less-frequently used functions. Furthermore, these complicated formulas may require experimentation to get right, and constructive complexity cannot capture a user’s intermediate missteps. We recognise these issues, and we do not mean to claim too much for this concept. The notion of constructive complexity knows nothing of a user’s familiarity with a given function.

4.5 Mathematical programming with Excel’s solver, O(1)

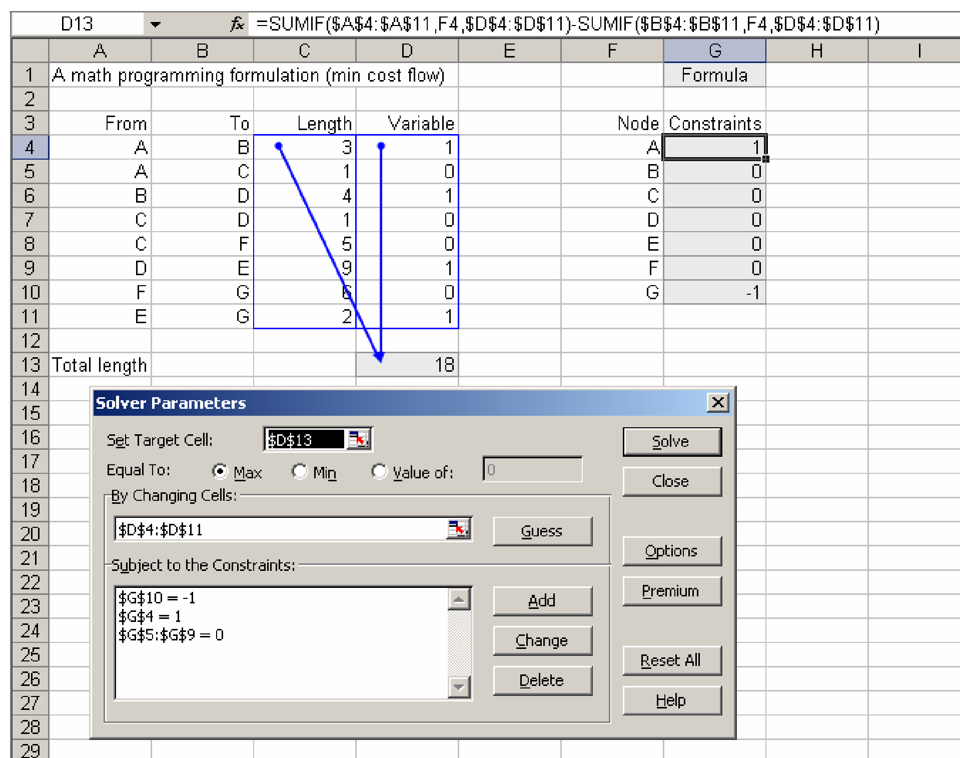
4.5.1 All constraints in Solver’s dialog box

The SPP is a classical application of network programming. In this case, the capacity of every arc is 1 and decision variables are either 0 or 1. The linear program can be given as minimise $\sum_{i,j} c_{ij} x_{ij}$, subject to $\sum_j x_{1,j} = -1$, $\sum_i x_{ij} = \sum_i x_{ji}$ for $j = 2, \ldots, n-1$, $\sum_j x_{j,n} = 1$, x_{ij}

binary. Due to the underlying bipartite graph structure of this linear program, Solver working on real numbers always finds an integral solution. So Solver can be used without explicitly setting decision variables as binary.

Each arc is associated with a decision variable, which is set to 1 if it belongs to the shortest path, 0 otherwise. Figure 6 presents a first implementation where all constraints are defined in the dialog box. The source is indicated by setting the constraint \$G\$4 = 1. The destination vertex is defined by setting the flow constraint \$G\$10 = –1. Flows are calculated using the difference between input flows and output flows. For each vertex B–F, net flow is set to 0 by setting the constraint \$G\$5:G\$9 = 0 in the dialog box. A similar version is presented in Winston and Albright (2000).

Figure 6 Spreadsheet model with constraints defined in the dialog box (see online version for colours)



Key cell formulas

Cell	*Formula*	*Copied to*
D13	= SUMPRODUCT(C4:C11,D4:D11)	–
G4	= SUMIF(\$A\$4:\$A\$11,F4,\$D\$4:\$D\$11) – SUMIF(\$B\$4:\$B\$11,F4, \$D\$4:\$D\$11)	G5:G10

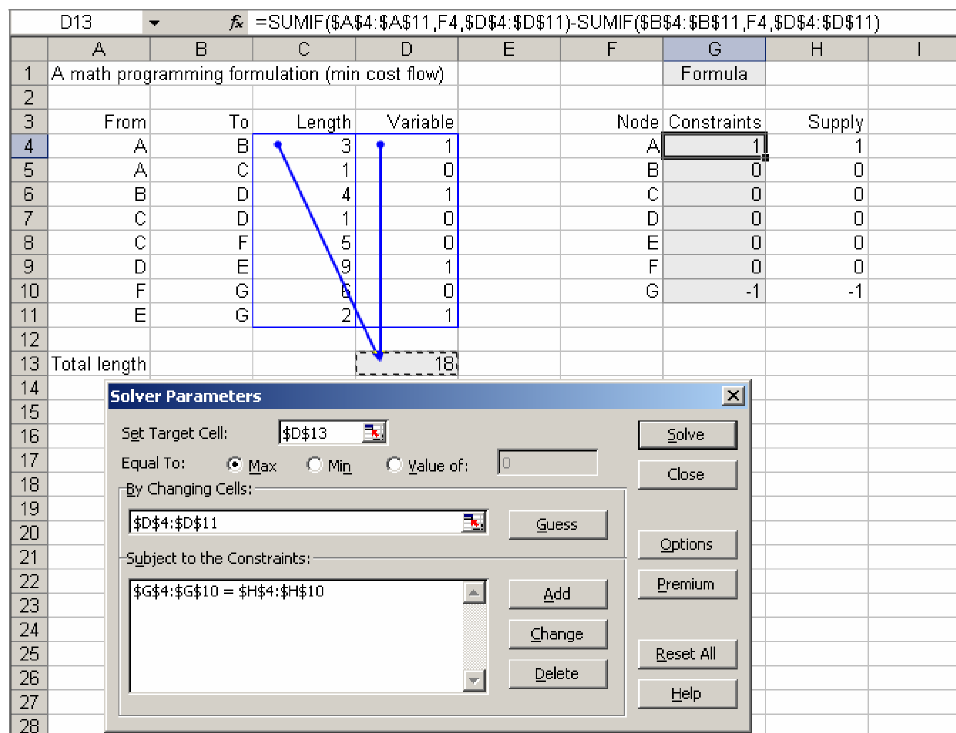
The length (in cell D13) of the shortest path is calculated as a SUMPRODUCT() of edge arc lengths and decision variables. Flows on nodes are calculated by the difference of the input flow and the output flow, calculated by two SUMIF functions. Then, Solver computes the minimal solution.

The constructive complexity is $O(1)$ since only two formulas need to be entered, whatever the network size. In addition, the revisional complexity is in $O(1)$, since it is always less than or equal to the constructive complexity. Nevertheless, when we want to change source or destination vertices, we need to change constraints in Solver's dialog boxes. Next, we give a revision of this model to avoid such a problem.

4.5.2 All constraints in the grid

Figure 7 presents a version in which the right hand sides of the constraints in the grid, and parameterised in Solver. We add the column supply that defines net flows on every node of the network. Such data are inputs of the model. In comparison with the previous version, changing the source node or the destination node does not require us to change constraints in the dialog box. In addition, this version has a constructive complexity in $O(1)$, but has a lower constructive complexity than the first example. In comparison with the first version, the second model is easier to understand and to extend. Such a version is presented in Ragsdale (2004).

Figure 7 Math programming approach: all constraint in the grid (see online version for colours)



Note: Key formulas are the same as in Figure 6.

The experienced operations researcher may see ways to improve our simple models. For example, consider the node information in columns F, G and H of Figure 5–7. If these blocks were moved below the 'From To' blocks, then new arcs and nodes could be inserted more easily. We set out these blocks horizontally to save vertical space for the purpose of printing here.

We could write models for Solver with the matrix structure of Figure 4, but will omit them for brevity. The point is that the way that Solver is used affects the constructive complexity, because Solver contains information about the model. Complexity is in fact embedded in Solver's dialog boxes. In general, Solver should not usually contain constants other than zero, just as a formula should not contain constants. Solver is often easier to use if constraints are defined with a right hand size of zero.

5 Extensions

It may be that constructive complexity should be measured more precisely. Our examples show that the order of magnitude test does not capture everything about constructive complexity. For example, a closer analysis of constructive complexity would eliminate spurious cells of the form $x = y$. As we have seen above, different spreadsheets can have constructive complexity of $O(1)$, but the spreadsheets can have different designs. Future work might distinguish spreadsheets with the same order of constructive complexity. Most likely, this means focusing on the number of distinct formulas. Minimising the number of distinct formulas raises a number of interesting issues.

First, reducing the number of cells tends to lengthen formulas. Compare, for example, the formulas in Figure 2 with those in Figure 5. On the other hand, in some corporate monstrosities we have seen, multiple formulas could be greatly simplified, resulting in much simpler models. We have seen bloated multi-sheet beasts simplified to a single screen, with modest formula lengths.

Second, some spreadsheet writers want to show intermediate values. The affinity for intermediate values can go overboard, where the writer assumes that the reader wants to see everything. But, the desire for intermediate values can be valid, so we need to consider clearly the needs of the reader.

Third, a tension remains between the spreadsheet writer and reader. With strict attention to constructive complexity, the writer may minimise the number of keystrokes require to produce the desired output. But, that keystroke sequence may produce a visual result that is more verbose or takes up more visual space than is desired by the reader, who is likely to prefer concise output.

So, the spreadsheet writer can misjudge by giving too much (such as unneeded intermediate formulas) or too little, among many other ways. Except for hiding constants in formulas, the over-concise spreadsheet is rare in our experience. Our preference is for conciseness (in the number of cells with formulas) as a key attribute. Obfuscation usually occurs by giving too much. Verbosity, according to the common wisdom, is a virtue in spreadsheets. (References cited earlier prescribed long lists of 'features' to include in every spreadsheet. See also Freeman (1996) and Mather (1999).) But in every art related to spreadsheets – writing, mathematics, graphics, math modelling – conciseness is considered a virtue.

Advanced features such as array functions, matrix operations and circular references can help improve constructive complexity. By extrapolation, the notion of constructive

complexity should help suggest new spreadsheet features. An operation that requires high constructive complexity motivates development of a feature to improve the spreadsheet writer's productivity. Similarly, a high constructive complexity can motivate construction of a macro to simplify a task, and an analysis of constructive complexity would show which macros would be of most help to the spreadsheet reader.

6 Conclusions

We have introduced the concept of constructive complexity, which allows a quantitative approach to studying the structure of a spreadsheet. Constructive complexity touches on a variety of computing issues, including spreadsheet auditing, perception, and the graphical relationship between an algorithm and its spreadsheet implementation. Constructive complexity can help show why one spreadsheet design is better than another, and thus can help rank different designs.

One could argue, however, that users rarely have the luxury of ranking spreadsheet designs for a given problem. It may be that calculating constructive complexity imposes too great of a burden on the spreadsheet writer. However, the writer will then probably be unaware of any number of issues regarding spreadsheet design. In short, if the writer cannot think through the structure of the spreadsheet in a quantitative way, we might worry about the writer's ability to produce a valid spreadsheet, which is inherently a quantitative task.

Constructive complexity does not clearly show how to improve a spreadsheet (just as computational complexity cannot clearly show how to improve an algorithm), though it can indicate what to improve. For example, the annoying keystrokes in Figure 2 virtually cried out for improvement. Efforts spent on finding better formulas are likely to also improve the probability that the spreadsheet is correct. Ideally, spreadsheet writers would engage in a friendly competition to produce simpler models, as our anonymous referees did. We omitted other versions of the shortest path spreadsheet, as colleagues and referees have given us improved versions, each more elegant than the previous. Constructive complexity is part of the scorecard.

Students should be invited to think about the number of keystrokes required to do a given task. If a part of a spreadsheet is clearly cumbersome to develop, that part will probably also be cumbersome to modify and audit. Just as a programmer may wish to optimise the slowest part of a computer program, a spreadsheet writer should be encouraged to think about which parts of the spreadsheet take the most time to write. Nevertheless, this first attempt at quantitative analysis of spreadsheet design may lead to related work, and possibly even automated tools.

Constructive complexity clearly omits other aspects of modelling. Except for the extra keystrokes, constructive complexity does not clearly touch on other aspects of spreadsheet design, such as colour or shading, range names or the likelihood that the client is familiar with a given function.

The main contribution here is to quantify one part of spreadsheet modelling, as a way of quantifying the effort in this type of problem solving. Clearly more work and debate are needed to better understand what makes for good spreadsheet design.

References

Bissell, J.L. (1986) 'Spreadsheet planning and design', *Journal of Accountancy*, Vol. 161, pp.110–120.

Bromley, R.G. (1985) 'Template design and review: how to prevent spreadsheet disasters', *Journal of Accountancy*, Vol. 160, pp.134–142.

Card, S.K., Moran, T.P. and Newell, A. (1983) *The Psychology of Human–Computer Interaction*. Hillsdale, NJ: L. Erlbaum Associates.

Conway, D.G. and Ragsdale, C.T. (1997) 'Modelling optimization problems in the unstructured world of spreadsheets', *Omega*, Vol. 25, pp.313–322.

Cragg, P.B. and King, M. (1993) 'Spreadsheet modelling abuse: an opportunity for OR?', *Journal of the Operational Research Society*, Vol. 44, pp.743–752.

Crain, J.L. and Fleenor, W.C. (1989) 'Standardizing spreadsheet designs', *CPA Journal*, Vol. 59, pp.81–84.

Dantzig, G. (1963) *Linear Programming and Extensions*. Princeton, NJ: Princeton University Press.

Dantzig, G., Fulkerson, R. and Johnson, S. (1954) 'Solution of a large-scale traveling-salesman problem', *Operations Research*, Vol. 2, p.393.

Davis, S.J. (1996) 'Tools for spreadsheet auditing', *Int. J. Human Computer Studies*, Vol. 45, pp.429–442.

Edge, W.R. and Wilson, E.J.G. (1990) 'Avoiding the hazards of microcomputer spreadsheets', *Internal Auditor*, Vol. 47, pp.35–39.

Freeman, D. (1996) 'How to make spreadsheets error-proof', *Journal of Accountancy*, Vol. 181, pp.75–77.

Johnson, T.E. (2007) *Spreadsheet Composer*, Available at: www.dustfreesolutions.com, Accessed 9 Oct 2007.

Jones, S.P., Blackwell, A and Burnett, M. (2003) 'A user-centred approach to functions in excel', *ACM SIGPLAN Notices*, Vol. 38, pp.165–176.

Kee, R. (1988) 'Programming standards for spreadsheet software', *CMA Mag*, Vol. 62, pp.55–60.

Kee, R.C. and Mason, J.O., Jr. (1988) 'Preventing errors in spreadsheets', *Internal Auditor*, Vol. 45, pp.42–47.

Mather, D. (1999) 'A framework for building spreadsheet based decision models', *Journal of the Operational Research Society*, Vol. 50, pp.70–74.

Nixon, D. and O'Hara, M. (2001) 'Spreadsheet auditing software', Paper presented in the Proceedings of the *European Symposium on Spreadsheet Risks*.

Paine, J. (2001) 'Ensuring spreadsheet integrity with model master', Paper presented in the Proceedings of the *EuSpRIG 2001*, Amsterdam 5–6 July.

Panko, R.R. and Sprague, R.H. Jr. (1998) 'Hitting the wall: errors in developing and code inspecting a 'simple' spreadsheet model', *Decision Support Systems*, Vol. 22, pp.337–353, April.

Powell, S.G. and Willemain, T.R. (2006) 'How novices formulate models. part I: qualitative insights and implications for teaching', *Journal of the Operational Research Society*, Vol. 58, pp.983–955.

Ragsdale, C.T. (2004) *Spreadsheet Modeling and Decision Analysis* (4th ed.). Boston, MA: Thompson-Southwestern Publishing.

Seal, K.C. (2001) 'A generalized PERT/CPM implementation in a spreadsheet', *INFORMS Transactions on Education*, Vol. 2, Available at: http://ite.informs.org/Vol2No1/seal/seal.html.

Stone, D.N. and Black, R.L. (1989) 'Using microcomputers: building structured spreadsheets', *Journal of Accountancy*. Vol. 168, pp.131–142.

Ward, S.C. (1989) 'Arguments for constructively simple models', *Journal of the Operational Research Society*, Vol. 40, pp.141–153.

Winston, W.L. and Albright, S.C. (2000) *Practical Management Science* (2nd ed.). Pacific Grove, CA: Duxbury.