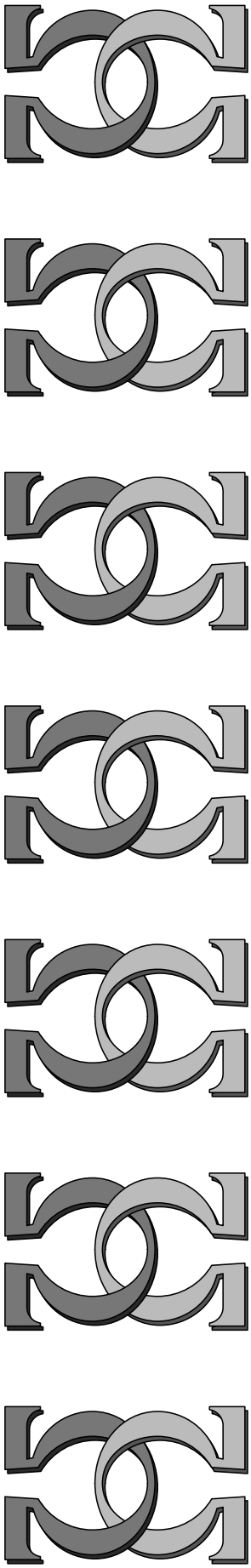

CDMTCS
Research
Report
Series

The Complexity of the Four Colour Theorem

Cristian S. Calude[1],
Elena Calude[2]

[1]University of Auckland, NZ
[2]Massey University at Albany, NZ

CDMTCS-368
August 2009; revised February 2010

Centre for Discrete Mathematics and
Theoretical Computer Science

The Complexity of the Four Colour Theorem

Cristian S. Calude, Elena Calude

To Professor S. Marcus on his 85th anniversary

Abstract

The four colour theorem states that the vertices of every planar graph can be coloured with at most four colours so that no two adjacent vertices receive the same colour. This theorem is famous for many reasons, including the fact that its original 1977 proof includes a non-trivial computer verification. Recently, a formal proof of the theorem was obtained with the equational logic program Coq [**14**].

In this paper we describe an implementation of the computational method introduced in [**7**, **5**] to evaluate the complexity of the four colour theorem. Our method uses a Diophantine equational representation of the theorem. We show that the four colour theorem is in the complexity class $\mathfrak{C}_{U,4}$. For comparison, the Riemann hypothesis is in $\mathfrak{C}_{U,2}$ while Fermat's last theorem is in class $\mathfrak{C}_{U,1}$.

1. *Introduction*

The four colour theorem—first conjectured in 1853 by Francis Guthrie—states that every plane separated into regions may be coloured using no more than four colours in such a way that no two adjacent regions receive the same colour. Two regions are called adjacent if they share a border segment, not just a point; regions must be contiguous, i.e. the plan has no exclaves.

In graph-theoretical terms, the four colour theorem states that the vertices of every planar graph can be coloured with at most four colours so that no two adjacent vertices receive the same colour. Shortly, every planar graph is four-colourable.

The theorem was proved in 1977 [**1**, **2**] (see also [**17**]) using a computer-assisted proof which consists in constructing a finite set of "configurations," and verifying that each of them is "reducible"—which implies that no configuration with this property can appear in a minimal counterexample to the theorem. Checking the correctness of the original proof is a very difficult task: it implies, among other things, checking the inputting of the descriptions of 1476 graphs, checking the correctness of the programs, proving the correctness of the compiler used to compile the programs, checking the degree of reliability of the hardware used to ran the programs.[†] Various partial independent verifications have been obtained[‡] culminating with

---

2000 *Mathematics Subject Classification* 03D15 (primary), 11Y16 (secondary).

This work was supported in part by The Andrea von Braun Foundation, Munich, under the grant for "Artistic Forms and Complexity."

[†]This computer-assisted proof generated much mathematical and philosophical discussions around the notion of acceptable mathematical proof, see for example [**3**, **8**, **9**].

[‡]It appears that there is no verification in its entirety.

the formal confirmation announced in [**16**] which uses the equational logic program Coq.[§] The following part of the concluding discussion in [**16**] is relevant for the current status of the proof:

> *However, an argument can be made that our 'proof' is not a proof in the traditional sense, because it contains steps that can never be verified by humans. In particular, we have not proved the correctness of the compiler we compiled our programs on, nor have we proved the infallibility of the hardware we ran our programs on. These have to be taken on faith, and are conceivably a source of error. ...*
>
> *However, from a practical point of view, the chance of a computer error that appears consistently in exactly the same way on all runs of our programs on all the compilers under all the operating systems that our programs run on is infinitesimally small compared to the chance of a human error during the same amount of case-checking.*
>
> *Apart from this hypothetical possibility of a computer consistently giving an incorrect answer, the rest of our proof can be verified in the same way as traditional mathematical proofs. We concede, however, that* verifying a computer program is much more difficult than checking a mathematical proof of the same length.[†]

The four colour property is mainly of mathematical interest: K. May, quoted by [**17**], p. 2, says that "Maps utilising only four colours are rare, and those that do usually require only three. Books on cartography and the history of mapmaking do not mention the four-colour property."

We describe an implementation of the computational method, motivated by its semiotic dimension [**15**], introduced in [**7**, **5**] to evaluate the complexity of the four colour theorem. Our method uses a Diophantine equational representation of the theorem. We show that the four colour theorem is in the complexity class $\mathfrak{C}_{U,4}$; the Riemann hypothesis and the Fermat last theorem are in the lower complexity classes $\mathfrak{C}_{U,2}$ and $\mathfrak{C}_{U,1}$, respectively.

2. *A diophantine equational representation of the four colour property*

We use the Diophantine representation of the four colour theorem proposed in [**13**], i.e. a Diophantine equation

$$F(n,t,a,\ldots)=0, \tag{2.1}$$

such that (2.1) has no solution if and only if every planar graph can be coloured with at most four colours so that no two adjacent vertices receive the same colour.

Actually, it is better to use a pre-Diophantine representation given by the following conditions. Without restricting the generality we consider all maps T_n consisting of the points (x,y) such that $J(x,y)\leq Q=(n^2+3n)/2$, where J is Cantor's bijection $J(x,y)=((x+y)^2+3x+y)/2$. Given a 4-colouring of T_n, $t_0,t_1,\ldots,t_Q$ there exist (and can be effectively computed) s,t such that for all $0\leq i\leq Q$:[‡]

$$t_i=rem(t,1+s(i+1)).$$

In other words, the sequence $t_0,t_1,\ldots,t_Q$ can be coded by s and t.

Every sequence $u_0,u_1,\ldots,u_Q$ with $u_i<4$ can be represented by some $u\leq R=(1+4(Q+2)!)^{Q+1}$ such that

$$u_i=rem(u,1+4(Q+2)!(i+1)).$$

[§]See [**14**] for a recent presentation of the formal proof.

[†]Our emphasis.

[‡]The integer remainder function is denoted by rem.

Finally, there is a map (say T_n) which cannot be coloured with 4 colours if and only if the following condition is satisfied:

$$(\exists n, t, s)(\forall u \leq R)(\exists x, y)(x + y \leq n)[A(x, y) \vee B(x, y)],$$

where

$$A(x, y) = u_{J(x,y)} \geq 4,$$

$$\begin{aligned} B(x, y) = [&(t_{J(x,y)} = t_{J(x+1,y)} \wedge u_{J(x,y)} \neq u_{J(x+1,y)}) \\ &\vee (t_{J(x,y)} \neq t_{J(x+1,y)} \wedge u_{J(x,y)} = u_{J(x+1,y)}) \\ &\vee (t_{J(x,y)} = t_{J(x,y+1)} \wedge u_{J(x,y)} \neq u_{J(x,y+1)}) \\ &\vee (t_{J(x,y)} \neq t_{J(x,y+1)} \wedge u_{J(x,y)} = u_{J(x,y+1)})]. \end{aligned}$$

A simple inspection shows that the above condition is computable, so the four colour theorem is of the form $\forall n P(n)$, where P is a computable predicate, i.e. a Π_1–problem.

3. *The method*

We use a fixed "universal formalism" for programs, more precisely, a universal self-delimiting Turing machine U [**4**]; the machine U will be fully described in the next section. The machine U has to be *minimal* in the sense that none of its instructions can be simulated by a program for U written with the remaining instructions.

To every Π_1–problem $\pi = \forall \sigma P(\sigma)$ we associate the algorithm $\Pi_P = \inf\{n : P(n) = \text{ false}\}$ which systematically searches for a counter-example for π. There are many programs (for U) which implement Π_P; without loss of generality, any such program will be denoted also by Π_P. Note that π is true iff $U(\Pi_P)$ never halts.

The complexity (with respect to U) of a Π_1–problem π is defined by the length of the smallest-length program (for U) Π_P—defined as above—where minimisation is calculated for all possible representations of π as $\pi = \forall n P(n)$:[†]

$$C_U(\pi) = \min\{|\Pi_P| : \pi = \forall n P(n)\}.$$

Because the complexity C_U is incomputable, we can work only with upper bounds for C_U. As the exact value of C_U is not important, following [**6**] we classify Π_1–problems into the following classes:

$$\mathfrak{C}_{U,n} = \{\pi : \pi \text{ is a } \Pi_1\text{–problem}, C_U(\pi) \leq n \text{ kbit}^{\ddagger}\}.$$

As the four colour theorem is a Π_1–problem, we choose a specific representation $4\text{CT} = \forall \sigma P(\sigma)$, and based on it we write the program $\Pi_{4\text{CT}}$, optimise it in length, and finally use the size of the program $|\Pi_{4\text{CT}}|$ as an upper bound on $C_U(4\text{CT})$. We found that the four colour theorem is in the class $\mathfrak{C}_{U,4}$.

[†]For C_U it is irrelevant whether π is known to be true or false. In particular, the program containing the single instruction halt is not a Π_P program, for any P.

[‡]A kilobit (kbit or kb) is equal to 2^{10} bits.

4. *A universal prefix-free binary Turing machine*

We briefly describe the syntax and the semantics of a register machine language which implements a (natural) minimal universal prefix-free binary Turing machine U; it is a refinement, constructed in [**6**], of the languages in [**12**, **10**, **7**].

Any register program (machine) uses a finite number of registers, each of which may contain an arbitrarily large non-negative integer.

By default, all registers, named with a string of lower or upper case letters, are initialised to 0. Instructions are labeled by default with 0,1,2,...

The register machine instructions are listed below. Note that in all cases R2 and R3 denote either a register or a non-negative integer, while R1 must be a register. When referring to R we use, depending upon the context, either the name of register R or the non-negative integer stored in R.

=R1,R2,R3

If the contents of R1 and R2 are equal, then the execution continues at the R3-th instruction of the program. If the contents of R1 and R2 are not equal, then execution continues with the next instruction in sequence. If the content of R3 is outside the scope of the program, then we have an illegal branch error.

&R1,R2

The contents of register R1 is replaced by R2.

+R1,R2

The contents of register R1 is replaced by the sum of the contents of R1 and R2.

!R1

One bit is read into the register R1, so the contents of R1 becomes either 0 or 1. Any attempt to read past the last data-bit results in a run-time error.

%

This is the last instruction for each register machine program before the input data. It halts the execution in two possible states: either successfully halts or it halts with an under-read error.

A *register machine program* consists of a finite list of labeled instructions from the above list, with the restriction that the halt instruction appears only once, as the last instruction of the list. The input data (a binary string) follows immediately after the halt instruction. A program not reading the whole data or attempting to read past the last data-bit results in a run-time error. Some programs (as the ones presented in this paper) have no input data; these programs cannot halt with an under-read error.

The instruction `=R,R,n` is used for the unconditional jump to the n-th instruction of the program. For Boolean data types we use integers 0 = `false` and 1 = `true`.

For longer programs it is convenient to distinguish between the main program and some sets of instructions called "routines" which perform specific tasks for another routine or the main program. The call and call-back of a routine are executed with unconditional jumps.

5. *Binary coding of programs*

To compute an upper bound on $C_U(4\text{CT})$ we need to compute the size in bits of the program $\Pi_{4\text{CT}}$, so we need to uniquely code in binary the programs for U. To this aim we use a prefix-free coding as follows.

The binary coding of special characters (instructions and comma) is the following (ε is the empty string):

special characters	code	special characters	code
,	ε	+	111
&	01	!	110
=	00	%	100

Table 1

For registers we use the prefix-free code $\text{code}_1 = \{0^{|x|}1x \mid x \in \{0,1\}^*\}$. Here are the codes of the first 32 registers:[†]

register	code_1	register	code_1	register	code_1	register	code_1
R_1	010	R_9	0001010	R_{17}	000010010	R_{25}	000011010
R_2	011	R_{10}	0001011	R_{18}	000010011	R_{26}	000011011
R_3	00100	R_{11}	0001100	R_{19}	000010100	R_{27}	000011100
R_4	00101	R_{12}	0001101	R_{20}	000010101	R_{28}	000011101
R_5	00110	R_{13}	0001110	R_{21}	000010110	R_{29}	000011110
R_6	00111	R_{14}	0001111	R_{22}	000010111	R_{30}	000011111
R_7	0001000	R_{15}	000010000	R_{23}	000011000	R_{31}	00000100000
R_8	0001001	R_{16}	000010001	R_{24}	000011001	R_{32}	00000100001

Table 2

For non-negative integers we use the prefix-free code $\text{code}_2 = \{1^{|x|}0x \mid x \in \{0,1\}^*\}$. Here are the codes of the first 16 non-negative integers:

integer	code_2	integer	code_2	integer	code_2	integer	code_2
0	100	4	11010	8	1110010	12	1110110
1	101	5	11011	9	1110011	13	1110111
2	11000	6	1110000	10	1110100	14	111100000
3	11001	7	1110001	11	1110101	15	111100001

Table 3

The instructions are coded by self-delimiting binary strings as follows:

(1) `&R1,R2` is coded in two different ways depending on R2:[‡]

$$01\text{code}_1(\text{R1})\text{code}_i(\text{R2}),$$

where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer.

(2) `+R1,R2` is coded in two different ways depending on R2:

$$111\text{code}_1(\text{R1})\text{code}_i(\text{R2}),$$

where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer.

[†]The register names are chosen to optimise the length of the program, i.e. the most frequent registers have the smallest code_1 length.

[‡]As $x\varepsilon = \varepsilon x = x$, for every string $x \in \{0,1\}^*$, in what follows we omit ε.

(3) `=R1,R2,R3` is coded in four different ways depending on the data types of R2 and R3:

$$00\text{code}_1(\text{R1})\text{code}_i(\text{R2})\text{code}_j(\text{R3}),$$

where $i = 1$ if R2 is a register and $i = 2$ if R2 is an integer, $j = 1$ if R3 is a register and $j = 2$ if R3 is an integer.

(4) `!R1` is coded by

$$110\text{code}_1(\text{R1}).$$

(5) `%` is coded by

$$100.$$

All codings for instruction names and special symbol comma, registers and non-negative integers are self-delimiting; the prefix-free codes used for registers and non-negative integers are disjoint. The code of any instruction is the concatenation of the codes of the instruction name and the codes (in order) of its components, hence the set of codes of instructions is prefix-free. The code of a program is the concatenation of the codes of its instructions, so the set of codes of all programs is prefix-free too.

The smallest program which halts is 100 and smallest program which never halts 00 010 010 100 100.

For example the following register machine routine computes in d the integer remainder of a divided by b, for integers $a \geq b \geq 0$ (if $b = 0$ then $d = 0$):[†]

instruction number	instruction	code	length
0	`&h,e`	01 0001001 00110	14
1	`&e,b`	01 00110 011	10
2	`&d,0`	01 00101 100	10
3	`=e,a,8`	00 00110 010 1110010	17
4	`+e,1`	111 00110 101	11
5	`+d,1`	111 00101 101	11
6	`=d,b,2`	00 00101 011 11000	15
7	`=a,a,3`	00 010 010 11001	13
8	`&e,h`	01 00110 0001001	14
9	`=a,a,c`	00 010 010 00100	13

Table 4

The routine can be uniquely encoded by concatenating the binary strings coding the instructions of the routine:

$$01000100100110010011001101001011000000110010111001011100110101 11$$

$$10010110100001010111100000010010110010100110000100100010010001 00$$

which is a string of length 128 bits.

[†]We use: R_1 = a, R_2 = b, R_3 = c, R_4 = d, R_5 = e, R_8 = h.

6. *The program* Π_{4CT}

In this section we present the program Π_{4CT} which is based on the computable predicate P described in section 2. In order to help the understanding the following formulas are used in the program:

- UFC(a) $= rem(u, 1 + 4(Q + 2)!(a + 1))$,
- TFC(a) $= rem(t, 1 + s(a + 1))$,
- CMP(a, b) $= \begin{cases} 1, & \text{if } a < b, \\ 0, & a = b, \\ 2, & a > b, \end{cases}$
- MUL(a,b) $= ab$,
- FAC(a) $= 1 \cdot 2 \cdot 3 \cdots a$,
- REM(a,b) $= rem(a, b)$,
- JFC(a,b) $= J(a, b)$.

```
 0 =a,a,116 //the main  program starts at instruction 116
 1 &A,a       //===UFC(a)
 2 &B,b       //copy a,b,c locally
 3 &C,c
 4 &a,q
 5 +a,2       //a =Q+2
 6 &c,8
 7 =a,a,76
 8 &a,d       //d = (Q+2)!
 9 +a,a
10 +a,a       //a = 4(Q+2)!
11 &c,15
12 &b,A
13 +b,1       //b = a+1
14 =a,a,50
15 &b,d
16 +b,1      //b = 1+4(Q+2)!(a+1)
17 &a,u
18 &c,20
19 =a,a,60
20 =d,1,27   //u < 1+4(Q+2)!(a+1)
21 &c,23     //u >= 1+4(Q+2)!(a+1)
22 =a,a,91   //d = REM(u,1+4(Q+2)!(a+1))
23 &a,A
24 &b,B
25 &c,C
26 =a,a,c
27 &d,a
28 =a,a,23
```

```
29 &aa,a      //===TFC(a)
30 &bb,b
31 &cc,c
32 +a,1       //a = a+1
33 &b,s       //b = s
34 &c,36
35 =a,a,50    //d =(a+1)s
36 &b,d
37 +b,1       //b = 1+(a+1)s
38 &a,t       //a = t
39 &c,41
40 =a,a,60    //d = CMP(t,1+(a+1)s)
41 =d,1,48    //t < 1+(a+1)s
42 &c,44      //t >= 1+s(a+1)
43 =a,a,91    //d = REM(t,1+s(a+1))
44 &a,aa
45 &b,bb
46 &c,cc
47 =a,a,c
48 &d,a       //d = t
49 =a,a,44
50 &ee,e      //===MUL(a,b)
51 &d,0
52 =b,0,58     //d = 0
53 &e,1
54 +d,a
55 =e,b,58     //d = ab
56 +e,1
57 =a,a, 54
58 &e,ee
59 =a,a,c
60 &ee,e       //===CMP(a,b)
61 &ff,f
62 &e,a
63 &f,b
64 +e,1
65 +f,1
66 &d,0
67 =e,f,73   //a = b
68 &d,1
69 =e,b,73   //a < b
70 &d,2
71 =f,a,73   //a > b
72 =a,a,64
73 &f,ff
74 &e,ee
75 =a,a,c
76 &aa,a     //FAC(a)
77 &bb,b
78 &cc,c
79 &b,0
```

```
80 &d,1
81 =b,aa,87 //d=a!
82 +b,1     //b < a
83 &a,d
84 &c,86
85 =a,a,50   //d = (b-1)!b
86 =a,a,81
87 &a,aa
88 &b,bb
89 &c,cc
90 =a,a,c
91 &ee,e     //REM(a,b)
92 &e,b
93 &d,0
94 =e,a,99   //d = REM(a,b)
95 +e,1
96 +d,1
97 =d,b,93
98 =a,a,94
99 &e,ee
100 =a,a,c
101 &ee,e    //===JFC(a,b)
102 &ff,f
103 &e,a
104 +e,b
105 &d,0     //case a = b = 0
106 =e,0,112
107 &f,1
108 &d,f
109 =e,f,112 //d = JFC(a,b)
110 +f,1
111 =a,a,108
112 +d,a    //add extra a
113 &e,ee
114 &f,ff
115 =a,a,c
116 &m,1     //===main program
117 &n,0     //n = N
118 &a,n     //a = N
119 &b,0     //b = 0
120 &c,122
121 =a,a,101  //d = JFC(N,0)
122 &q,d     //q = JFC(N,0)
123 &a,q
124 +a,2     //a = q+2
125 &c,127
126 =a,a,76  //d = (q+2)!
127 &a,d     //a = (q+2)!
128 +a,a
129 +a,a
130 +a,1      //a = 1+ 4(q+2)!
```

```
131 &e,q
132 +e,1
133 &f,1
134 &r,a
135 =f,e,142  //r =(1+4(q+2)!)^(q+1)
136 &c,139    //f < e
137 &b,r      //b = r
138 =a,a,50
139 &r,d      //r = a^(f+1)
140 +f,1
141 =a,a,135
142 &t,0
143 &s,0
144 &u,0
145 +r,1      //r = R+1
146 =u,r,220  //u > R
147 &x,0      //u <= R
148 &y,0
149 &e,x
150 +e,y      //e = x+y
151 &r,n
152 +r,1      //R = N+1
153 =e,r,207  //x+y > N
154 &a,x      //x+y <= N, compute A(x,y) and B(x,y)
155 &b,y
156 &c,158
157 =a,a,101
158 &f,d      //f = JFC(x,y)
159 &e,x
160 +e,y
161 +e,1
162 +e,f      //e = JFC(x,y+1)
163 &g,e
164 +g,1      //g = JFC(x+1,y)
165 &a,f      //a = JFC(x,y)
166 &c,168
167 =a,a,1
168 &h,d      //h = u(JFC(x,y))
169 &c,171
170 =a,a,29   //d = t(JFC(x,y))
171 &k,d      //k = t(JFC(x,y))
172 &a,h      //a = u(JFC(x,y))
173 &b,4
174 &c,176
175 =a,a,60   //d = CMP(u(JFC(x,y)),4)
176 =d,1,179  //A(x,y) = false
177 +u,1      //A(x,y) = true
178 =a,a,146
179 &c,182
180 &a,g     //a = JFC(x+1,y)
181 =a,a,1
```

```
182 &i,d     //i = u(JFC(x+1,y))
183 &c,185
184 =a,a,29
185 &j,d     //j = t(JFC(x+1,y))
186 &c,189
187 &a,e     //a = JFC(x,y+1)
188 =a,a,1
189 &f,d     //f = u(JFC(x,y+1))
190 &c,192
191 =a,a,29
192 &g,d     //g = t(JFC(x,y+1))
193 =k,j,199
194 =h,i,177 //B(x,y) = true
195 =k,g,198
196 =h,f,177 //B(x,y) = true
197 =a,a,200 //B(x,y) = false
198 =h,f,200 //B(x,y) = false
199 =h,i,195
200 +y,1
201 &e,x
202 +e,y      //e = x+y
203 =e,r,205  //x+y > N
204 =a,a,155  //x+y <= N
205 +x,1
206 =a,a,148
207 +s,1
208 &e,m
209 +e,1
210 =s,e,212  //s > m
211 =a,a,144  //s <= m
212 +t,1
213 =t,e,215  //t > m
214 =a,a,143  //t <= m
215 +n,1
216 =n,e,218  //N > m
217 =a,a,118  //N <= m
218 +m,1
219 =a,a,117
220 %         //stop, 4CTH is false
```

We use the following conversion table for registers:

$R_1 = a$	$R_2 = e$	$R_3 = c$	$R_4 = d$	$R_5 = b$	$R_6 = f$	$R_7 = r$
$R_8 = ee$	$R_9 = h$	$R_{10} = x$	$R_{11} = y$	$R_{12} = g$	$R_{13} = n$	$R_{14} = aa$
$R_{15} = q$	$R_{16} = s$	$R_{17} = t$	$R_{18} = u$	$R_{19} = bb$	$R_{20} = cc$	$R_{21} = ff$
$R_{22} = i$	$R_{23} = k$	$R_{24} = m$	$R_{25} = A$	$R_{26} = j$	$R_{27} = B$	$R_{28} = C$

Consequently, the upper bound given by the size of the program Π_{4CT} for U is 3,489 bits, so the four colour theorem belongs to the class $\mathfrak{C}_{U,4}$.

7. *Final comments*

We have shown that the four colour theorem is in $\mathfrak{C}_{U,4}$. It is possible to decrease the size of the program $\Pi_{4\mathrm{CT}}$ by optimising the code or by using a different computable predicate. We conjecture that the four colour theorem is not in $\mathfrak{C}_{U,2}$.

The following mathematical statements are in lower complexity classes: Legendre's conjecture, Fermat's last theorem and Goldbach's conjecture are in $\mathfrak{C}_{U,1}$ and the Riemann hypothesis is in $\mathfrak{C}_{U,3}$ [**6**, **11**].

Acknowledgement

We thank Michael Dinneen and Nadia Kasto for critical comments and extensive discussions which improved this paper.

References

1. K. Appel, W. Haken, J. Koch. Every planar map is four colourable, I: Discharging, *Illinois J. Math.* 21 (1977), 429–490.
2. K. Appel, W. Haken. Every planar map is four-colorable, II: Reducibility, *Illinois J. Math.* 21 (1977), 491–567.
3. A. S. Calude. The journey of the four colour theorem through time, *The NZ Math. Magazine* 38, 3 (2001), 2735.
4. C. S. Calude. *Information and Randomness: An Algorithmic Perspective*, 2nd Edition, Revised and Extended, Springer-Verlag, Berlin, 2002.
5. C. S. Calude, E. Calude. Evaluating the Complexity of Mathematical Problems. Part 1, *Complex Systems* 18 (2009), 267–285.
6. C. S. Calude, E. Calude. Evaluating the complexity of mathematical problems. Part 2, *Complex Systems* 18 (2010), 387–401.
7. C. S. Calude, E. Calude, M. J. Dinneen. A new measure of the difficulty of problems, *Journal for Multiple-Valued Logic and Soft Computing* 12 (2006), 285–307.
8. C. S. Calude, E. Calude, S. Marcus. Passages of proof, *Bull. EATCS* 84 (2004), 167–188.
9. C. S. Calude, E. Calude, S. Marcus. Proving and programming, in C. S. Calude (ed.). *Randomness & Complexity, from Leibniz to Chaitin*, World Scientific, Singapore, 2007, 310–321.
10. C. S. Calude, M. J. Dinneen, C.-K. Shu. Computing a glimpse of randomness, *Experimental Mathematics* 11, 2 (2002), 369–378.
11. E. Calude. The Complexity of the Goldbach's Conjecture and Riemann's Hypothesis, *CDMTCS Research Report* 369, 2009, 14 pp.
12. G. J. Chaitin. *Algorithmic Information Theory*, Cambridge University Press, Cambridge, 1987. (third printing 1990)
13. M. Davis, Y. V. Matijasevič, J. Robinson. Hilbert's tenth problem. Diophantine equations: Positive aspects of a negative solution, in F. E. Browder (ed.). *Mathematical Developments Arising from Hilbert Problems*, American Mathematical Society, Providence, RI, 1976, 323–378.
14. G. Gonthier. Formal proof—the four color theorem, *Notices of AMS* 55, 11 (2008), 1382–1393.
15. S. Marcus. Mathematics through the glasses of Hjelmslev's semiotics, *Semiotica* 145-1/4 (2003), 235–246.
16. N. Robertson, D. P. Sanders, P. Seymour, R. Thomas. The Four Color Theorem, `http://www.math.gatech.edu/~thomas/FC/fourcolor.html` (accessed on 30 November 2008).
17. R. Wilson. *Four Colours Suffice*, Penguin, London, 2002.

C. S. Calude
Department of Computer Science
The University of Auckland
Private Bag 92019, Auckland, New Zealand
`www.cs.auckland.ac.nz/~cristian`

E. Calude
Institute of Information and Mathematical Sciences
Massey University at Albany
Private Bag 102-904, North Shore MSC
New Zealand
`http://www.massey.ac.nz/~ecalude`