NEWLY AVAILABLE SECTION OF
THE CLASSIC WORK

The Art of Computer Programming

VOLUME 4

Satisfiability

FASCICLE 6

DONALD E. KNUTH

CONTENTS

I can't get no ...

— MICK JAGGER and KEITH RICHARDS, *Satisfaction* (1965)

7.2.2.2. Satisfiability. We turn now to one of the most fundamental problems of computer science: Given a Boolean formula $F(x_1, \ldots, x_n)$, expressed in so-called "conjunctive normal form" as an AND of ORs, can we "satisfy" F by assigning values to its variables in such a way that $F(x_1, \ldots, x_n) = 1$? For example, the formula

$$F(x_1, x_2, x_3) = (x_1 \vee \bar{x}_2) \wedge (x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \qquad (1)$$

is satisfied when $x_1x_2x_3 = 001$. But if we rule that solution out, by defining

$$G(x_1, x_2, x_3) = F(x_1, x_2, x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3), \qquad (2)$$

then G is unsatisfiable: It has no satisfying assignment.

Section 7.1.1 discussed the embarrassing fact that nobody has ever been able to come up with an efficient algorithm to solve the general satisfiability problem, in the sense that the satisfiability of any given formula of size N could be decided in $N^{O(1)}$ steps. Indeed, the famous unsolved question "does P = NP?"

Another problem equivalent to satisfiability is obtained by going back to the Boolean interpretation in (1) and complementing both sides of the equation. By De Morgan's laws 7.1.1–(11) and (12) we have

$$\overline{F}(x_1, x_2, x_3) = (\bar{x}_1 \wedge x_2) \vee (\bar{x}_2 \wedge \bar{x}_3) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_2 \wedge \bar{x}_3); \qquad (5)$$

and F is unsatisfiable $\iff F = 0 \iff \overline{F} = 1 \iff \overline{F}$ is a tautology. Consequently F is satisfiable if and only if $\overline{F}$ is not a tautology: The tautology problem and the satisfiability problem are essentially the same.*

Since the satisfiability problem is so important, we simply call it SAT. And instances of the problem such as (1), in which there are no clauses of length greater than 3, are called 3SAT. In general, kSAT is the satisfiability problem restricted to instances where no clause has more than k literals.

---

* Strictly speaking, TAUT is coNP-complete, while SAT is NP-complete; see Section 7.9.

The method of showing a statement to be tautologous consists merely of constructing a table under it in the usual way and observing that the column under the main connective is composed entirely of 'T's.

— W. V. O. QUINE, *Mathematical Logic* (1940)

A Computing Procedure for Quantification Theory*

MARTIN DAVIS

Rensselaer Polytechnic Institute, Hartford Division, East Windsor Hill, Conn.

AND

HILARY PUTNAM

Princeton University, Princeton, New Jersey

The hope that mathematical methods employed in the investigation of formal logic would lead to purely computational methods for obtaining mathematical theorems goes back to Leibniz and has been revived by Peano around the turn of the century and by Hilbert's school in the 1920's. Hilbert, noting that all of classical mathematics could be formalized within quantification theory, declared that the problem of finding an algorithm for determining whether or not a given formula of quantification theory is valid was the central problem of mathematical logic. And indeed, at one time it seemed as if investigations of this "decision" problem were on the verge of success. However, it was shown by Church and by Turing that such an algorithm can not exist. This result led to consider-

A Machine Program for Theorem-Proving†

Martin Davis, George Logemann, and Donald Loveland

Institute of Mathematical Sciences, New York University

The programming of a proof procedure is discussed in connection with trial runs and possible improvements.

In [1] is set forth an algorithm for proving theorems of quantification theory which is an improvement in certain respects over previously available algorithms such as that of [2]. The present paper deals with the programming of the algorithm of [1] for the New York University, Institute of Mathematical Sciences' IBM 704 computer, with some modifications in the algorithm suggested by this work, with the results obtained using the completed algorithm. Familiarity with [1] is assumed throughout.

Changes in the Algorithm and Programming Techniques Used

The algorithm of [1] consists of two interlocking parts. The first part, called the *QFl-Generator*, generates (from the formula whose proof is being attempted) a growing propositional calculus formula in conjunctive normal form, the "quantifier-free lines." The second part, the *Processor*, tests, at regular stages in its "growth," the consistency of this propositional calculus formula. An inconsistent set

III*. *Splitting Rule* the form

$(A \vee$

where A, B, and R are and only if A & R an

Justification of R for $p = 1$, $F = B$ &

The forms of Rule theoretically they are each has certain desira cause of the fact tha number and the lengt rather quickly after s tive in a computer if Also, it was observe many duplicated and Some success was obta tematically eliminate total length increasin complicated problems III can seldom yield of Rule III* often wil

In programming storage. The rest of t out using only fast a Rule" is used one of on tape. Tape memor terial stack-of-plates the first to be read.

In the program writ

The Complexity of Theorem-Proving Procedures

Stephen A. Cook

University of Toronto

Summary

It is shown that any recognition problem solved by a polynomial time-bounded nondeterministic Turing machine can be "reduced" to the problem of determining whether a given propositional formula is a tautology. Here "reduced" means, roughly speaking, that the first problem can be solved deterministically in polynomial time provided an oracle is available for solving the second. From this notion of reducible, polynomial degrees of difficulty are defined, and it is shown that the problem of determining tautologyhood has the same polynomial degree as the

certain recursive set of strings on this alphabet, and we are interested in the problem of finding a good lower bound on its possible recognition times. We provide no such lower bound here, but theorem 1 will give evidence that {tautologies} is a difficult set to recognize, since many apparently difficult problems can be reduced to determining tautologyhood. By reduced we mean, roughly speaking, that if tautologyhood could be decided instantly (by an "oracle") then these problems could be decided in polynomial time. In order to make this notion precise, we introduce query machines, which are like Turing machines with oracles

Stephen Cook's Method I, from his Spring 1972 course notes, University of Toronto

It is suggested by Robinson's lemma and its proof follows from it.

It operates on an expanding set of clauses $C_1, C_2, \ldots, C_n$ (starting with the initial ones)

It has a push down stack of literals $\ell_1, \ldots, \ell_m$, consistent with no repetitions.

At every time we have a partial truth assignment obtained by assigning true to $\ell_1, \ldots, \ell_m$ on the stack. We think of the objective as finding a satisfying assignment.

1) $m \leftarrow 0$; $C_1, \ldots, C_n$ given (we start with initial clauses $C_1 \ldots, C_n$ and empty stack)

Comment:

At every time before we enter step 2) no clause is completely falsified by the partial truth assignment which verifies all $\ell_1, \ldots, \ell_m$.

2) $m \leftarrow m+1$

Select a literal ℓ_m such that neither ℓ_m nor $\overline{\ell}_m$ appears on the stack.

If no such ℓ_m exists then we have found a satisfying truth assignment, so $C_1, \ldots, C_n$ are consistent.

If no clause is falsified by verifying $\ell_1, \ldots \ell_m$, go to step 2).

If some clause is falsified go to step 3).

3) (Assume C_i is falsified when $\ell_1, \ldots, \ell_m$ are verified)

Replace ℓ_m by $\overline{\ell}_m$ on the stack.

An Empirical Comparison of Backtracking Algorithms

CYNTHIA A. BROWN AND PAUL W. PURDOM, JR.

Abstract—**In this paper we report the results of experimental studies of zero-level, one-level, and two-level search rearrangement backtracking. We establish upper and lower limits for the size problem for which one-level backtracking is preferred over zero-level and two-level methods, thereby showing that the zero-level method is best for very small problems. The one-level method is best for moderate size problems, and the two-level method is best for extremely large problems. Together with our theoretical asymptotic formulas, these measurements provide a useful guide for selecting the best search rearrangement method for a particular problem.**

Index Terms—**Backtracking, constraint satisfaction, search algorithms, tree search.**

I. INTRODUCTION

An important task of computer scientists is devising general algorithms that can be used to solve any problem from a large set of related problems. Such sets of problems can be divided into two classes, sometimes called "easy" and "hard." The easy sets are those for which each problem in the set can be solved

Mathematics People

Computer-Aided Verification Prize Awarded

The 2009 Computer-Aided Verification (CAV) award has been presented to the following seven individuals: CONOR F. MADIGAN, Kateeva, Inc.; SHARAD MALIK, Princeton University; JOAO P. MARQUES-SILVA, University College Dublin, Ireland; MATTHEW W. MOSKEWICZ, University of California Berkeley; KAREM A. SAKALLAH, University of Michigan; LINTAO ZHANG, Microsoft Research, and YING ZHAO, Wuxi Capital Group. They were honored for fundamental contributions to the development of high-performance Boolean satisfiability solvers.

The award recipients worked in two different teams, one at the University of Michigan and one at Princeton University, where they created powerful programs for checking whether a logic formula has a consistent solution. This is known as a "Boolean satisfiability problem". Satisfiability, or SAT, solvers can be used to solve a number of different problems, and it must be determined whether there is any way of satisfying all of them. SAT solvers have had a profound impact on the field of computer-aided verification, which is dedicated to the creation of tools that allow hardware and software designers to detect possible flaws in their systems and programs. Sakallah and Marques-Silva's GRASP solver, developed at the University

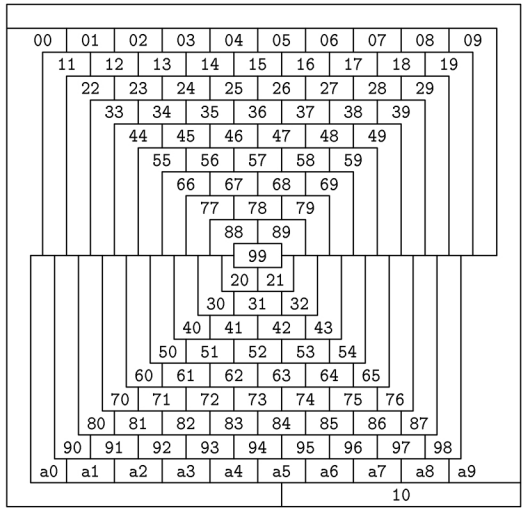
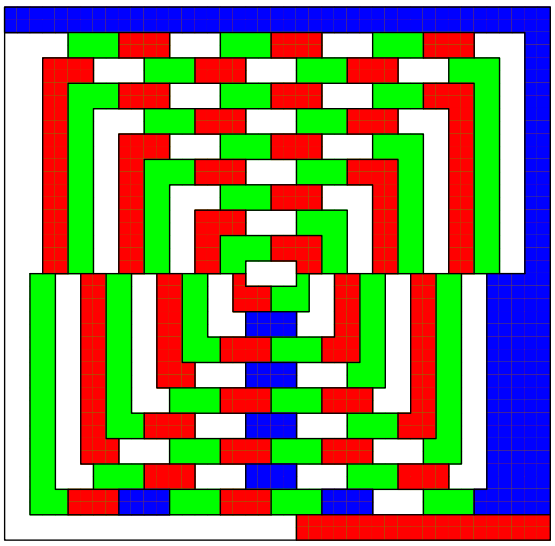
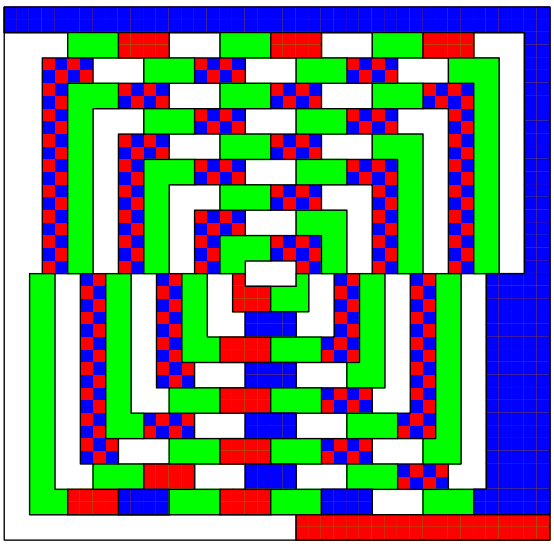


Fig. 33. The McGregor graph of order 10. Each region of this "map" is identified by a two-digit hexadecimal code. Can you color the regions with four colors, never using the same color for two adjacent regions?

Martin Gardner astonished the world in 1975 when he reported [*Scientific American* **232**, 4 (April 1975), 126–130] that a proper coloring of the planar map in Fig. 33 requires *five* distinct colors, thereby disproving the longstanding four-color conjecture. (In that same column he also cited several other "facts" supposedly discovered in 1974: (i) $e^{\pi\sqrt{163}}$ is an integer; (ii) pawn-to-king-rook-4 ('h4') is a winning first move in chess; (iii) the theory of special relativity is fatally flawed; (iv) Leonardo da Vinci invented the flush toilet; and (v) Robert Ripoff devised a motor that is powered entirely by psychic energy. Thousands of readers failed to notice that they had been April Fooled!)

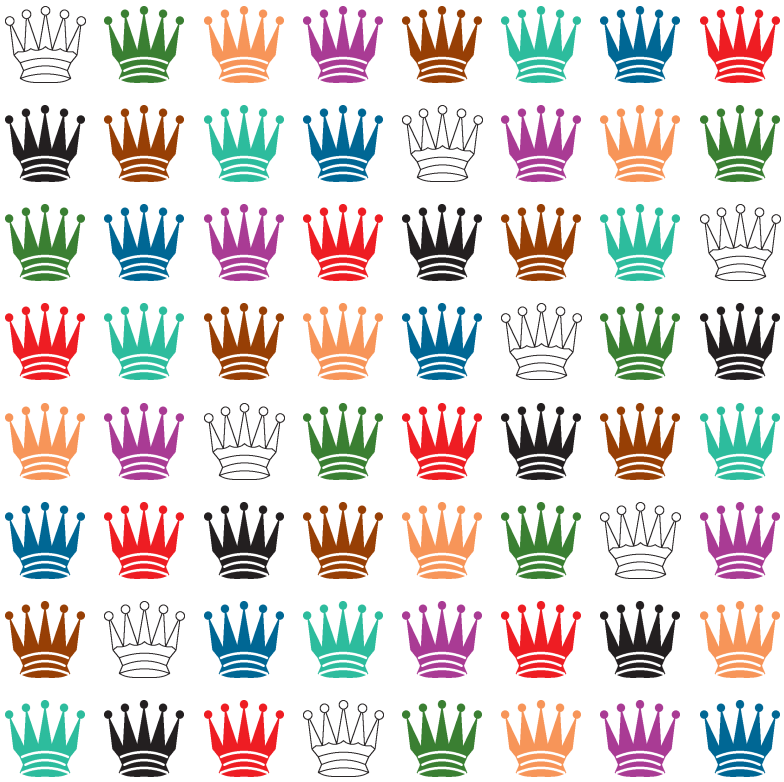


One color used only seven times!
(Randal Bryant)

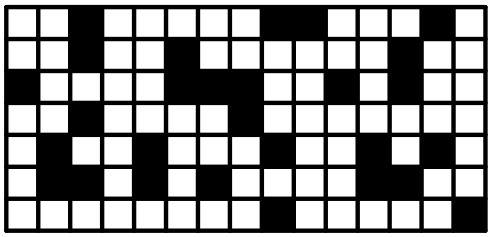


$2^{23} = 8{,}388{,}608$ ways to color!
(Frank Bernhart)

(The total number of ways actually turns out to be 898,431,907,970,211.)



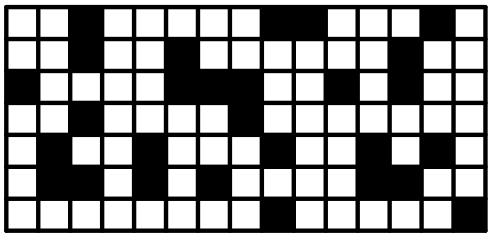
Conway's Game of Life



A live (black) cell stays alive if and only if two or three of its neighbors are alive.

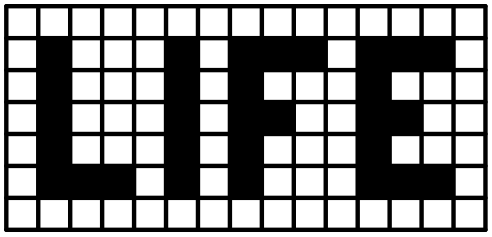
A dead (white) cell comes to life if and only if exactly three of its neighbors are alive.

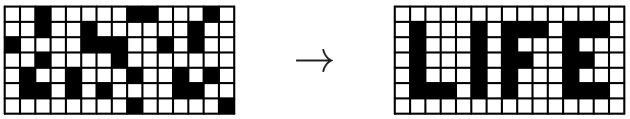
Conway's Game of Life

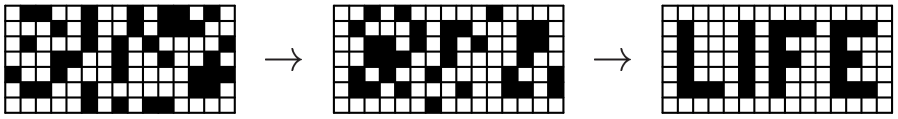


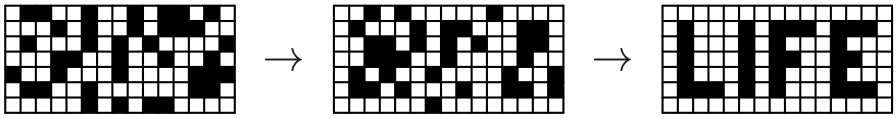
A live (black) cell stays alive if and only if two or three of its neighbors are alive.

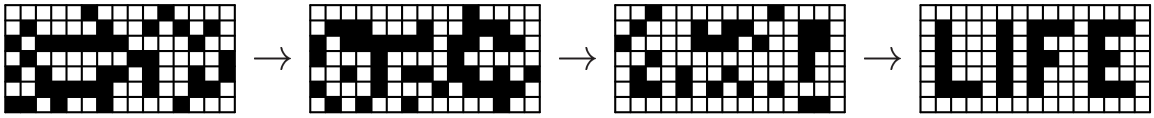
A dead (white) cell comes to life if and only if exactly three of its neighbors are alive.







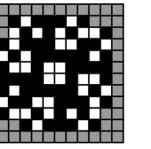




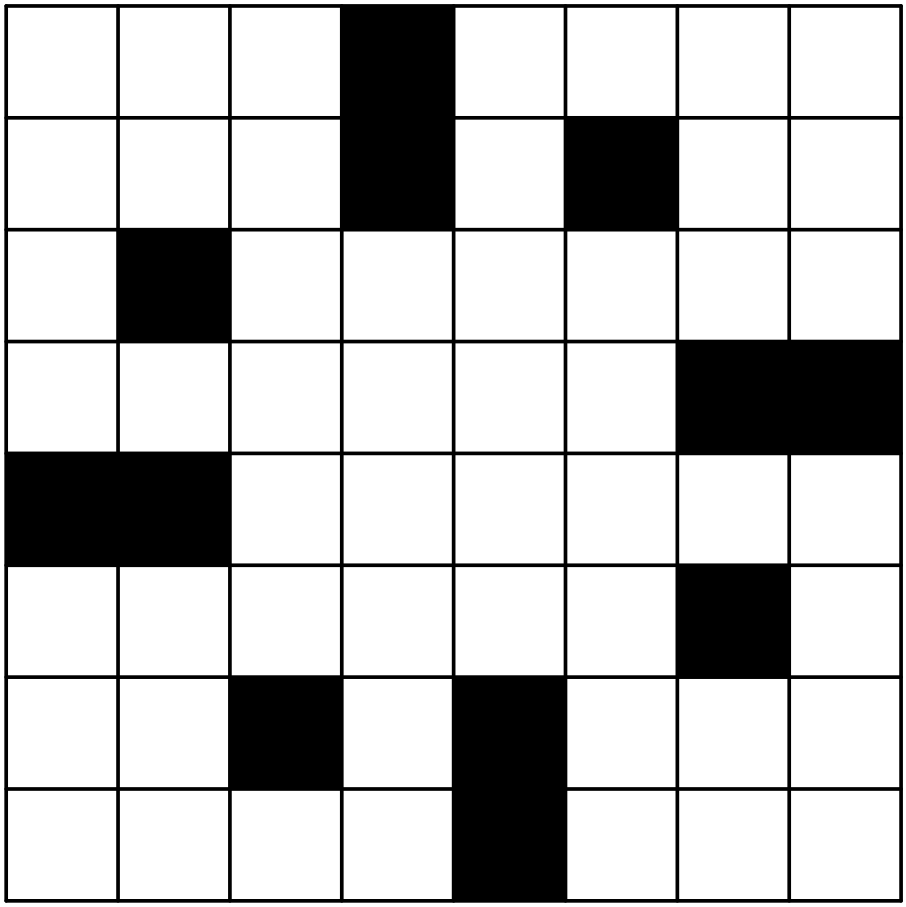
▸ **85.** [*39*] A *Garden of Eden* is a state of Life that has no predecessor.

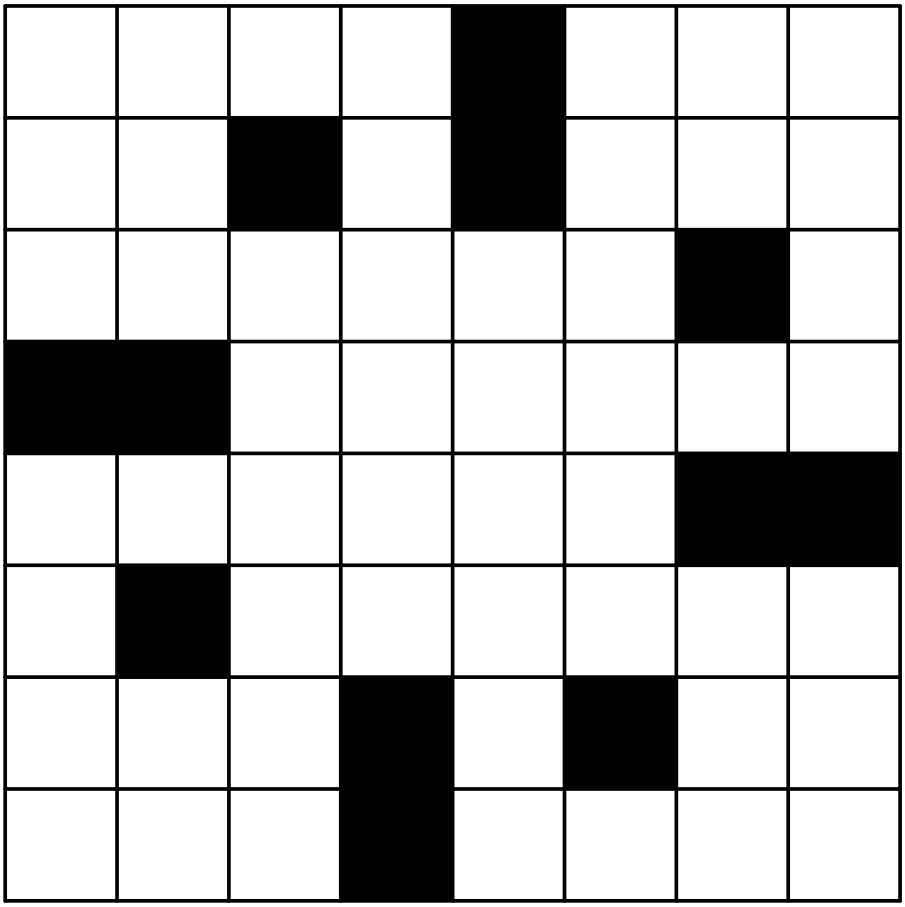
a) If the pattern of 92 cells illustrated here occurs anywhere within a bitmap X, verify that X is a Garden of Eden. (The gray cells can be either dead or alive.)

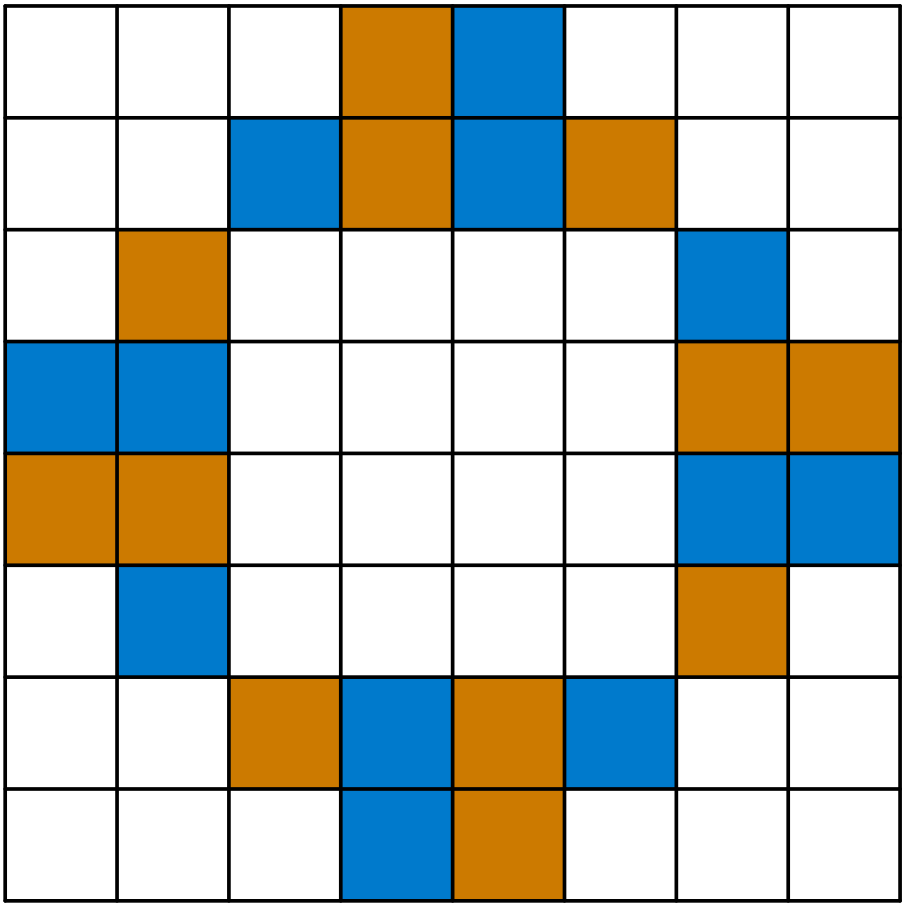
b) This "orphan" pattern, found with a SAT solver's help, is the smallest that is currently known. Can you imagine how it was discovered?

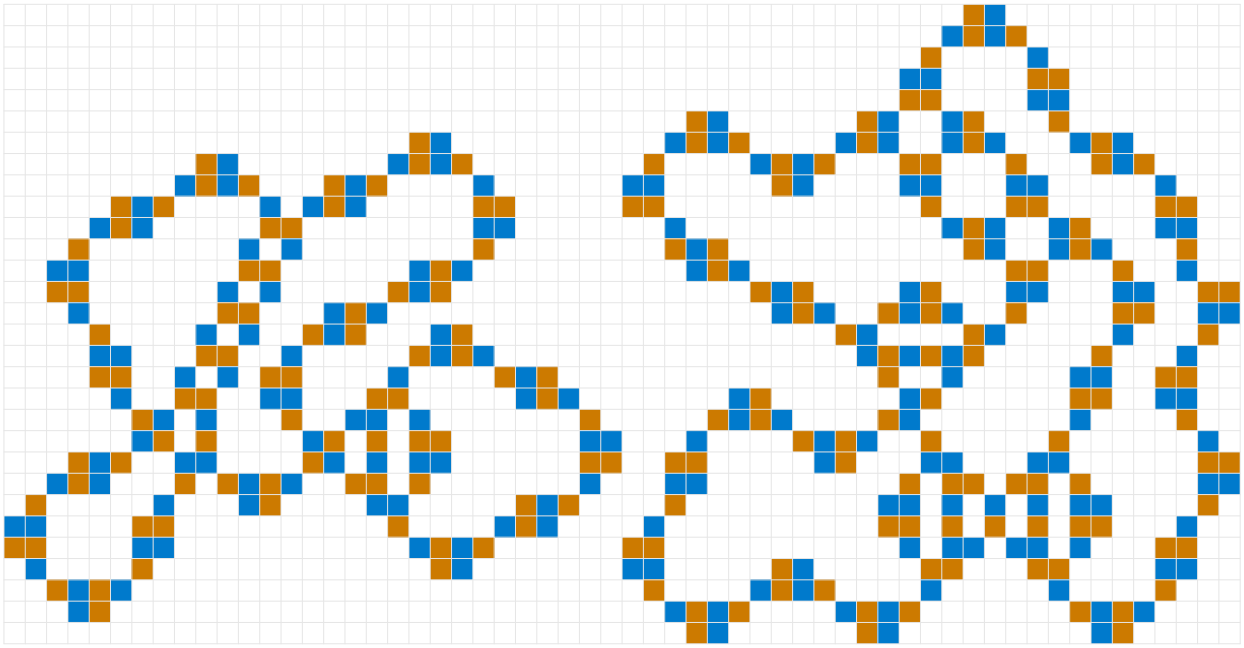


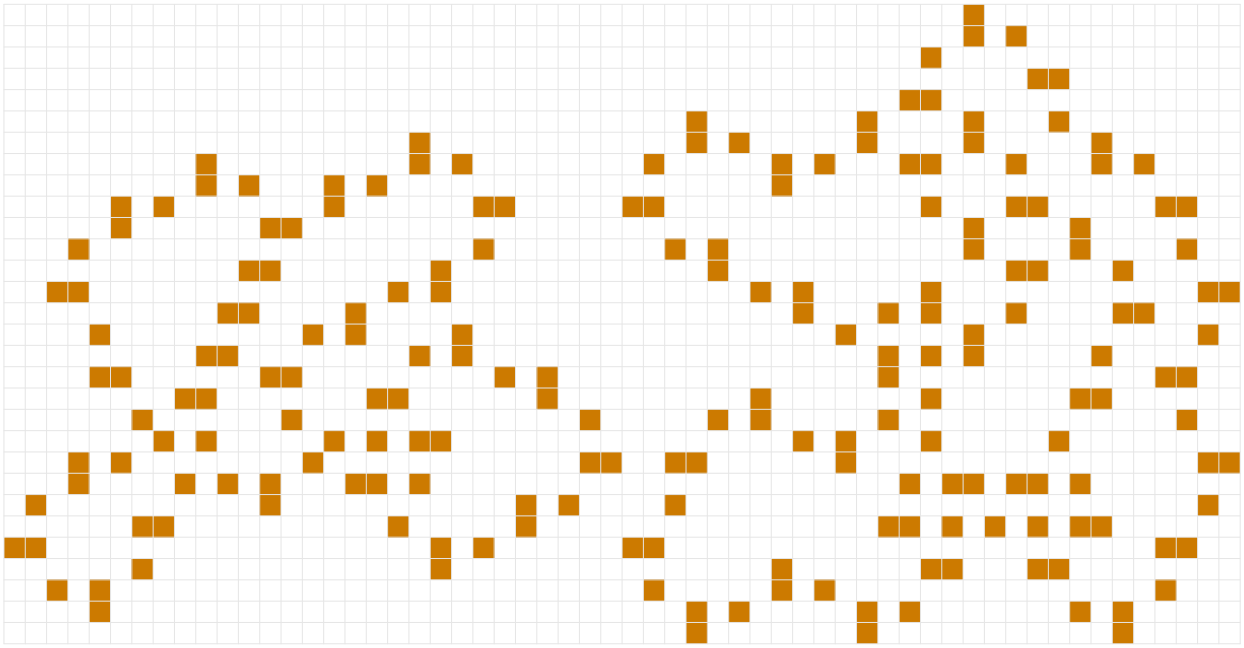
86. [*M23*] How many Life predecessors does a random 10×10 bitmap have, on average?

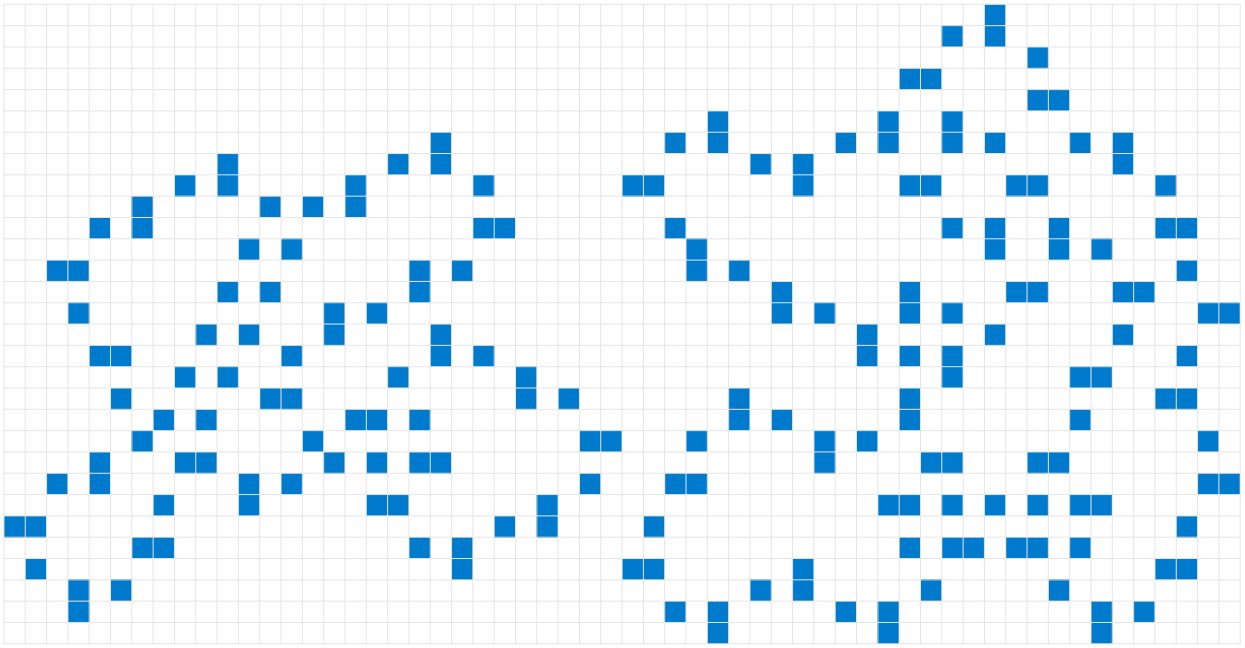












SAT0

November 5, 2012 at 14:01

1. Intro. This program is part of a series of "SAT-solvers" that I'm putting together for my own education as I prepare to write Section 7.2.2.2 of *The Art of Computer Programming*. My intent is to have a variety of compatible programs on which I can run experiments to learn how different approaches work in practice.

Indeed, this is the first of the series — more precisely the zero-th. I've tried to write it as a primitive baseline against which I'll be able to measure various technical improvements that have been discovered in recent years. This version represents what I think I would have written in the 1960s, when I knew how to do basic backtracking with classical data structures (but very little else). I have intentionally written it *before* having read *any* of the literature about modern SAT-solving techniques; in other words I'm starting with a personal "tabula rasa." My plan is to write new versions as I read the literature, in more-or-less historical order. The only thing that currently distinguishes me from a programmer of forty years ago, SAT-solving-wise, is the knowledge that better methods almost surely do exist.

[*Note:* The present code is slightly modified from the original SAT0. It now corresponds to what has become Algorithm 7.2.2.2A, so that I can make the quantitative experiments recorded in the book.]

Although this is the zero-level program, I'm taking care to adopt conventions for input and output that will be essentially the same in all of the fancier versions that are to come.

The input on *stdin* is a series of lines with one clause per line. Each clause is a sequence of literals separated by spaces. Each literal is a sequence of one to eight ASCII characters between `!` and `}`, inclusive, not beginning with `~`, optionally preceded by `~` (which makes the literal "negative"). For example, Rivest's famous clauses on four variables, found in 6.5–(13) and 7.1.1–(32) of *TAOCP*, can be represented by the following eight lines of input:

```
x2 x3 ~x4
x1 x3 x4
~x1 x2 x4
~x1 ~x2 x3
~x2 ~x3 x4
~x1 ~x3 ~x4
x1 ~x2 ~x4
x1 x2 ~x3
```

Input lines that begin with `~␣` are ignored (treated as comments). The output will be '`~`' if the input clauses are unsatisfiable. Otherwise it will be a list of noncontradictory literals that cover each clause, separated by spaces. ("Noncontradictory" means that we don't have both a literal and its negation.) The input above would, for example, yield '`~`'; but if the final clause were omitted, the output would be '`~x1 ~x2 x3`', in some order, possibly together with either `x4` or `~x4` (but not both). No attempt is made to find all solutions; at most one solution is given.

The running time in "mems" is also reported, together with the approximate number of bytes needed for data storage. One "mem" essentially means a memory access to a 64-bit word. (These totals don't include the time or space needed to parse the input or to format the output.)

2. So here's the structure of the program. (Skip ahead if you are impatient to see the interesting stuff.)

```
#define o mems++ /* count one mem */
#define oo mems += 2 /* count two mems */
#define ooo mems += 3 /* count three mems */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_flip.h"
  typedef unsigned int uint; /* a convenient abbreviation */
  typedef unsigned long long ullng; /* ditto */
  ⟨ Type definitions 5 ⟩;
  ⟨ Global variables 3 ⟩;
  ⟨ Subroutines 27 ⟩;
  main(int argc, char *argv[])
  {
    register uint c, h, i, j, k, l, p, q, r, level, parity;
    ⟨ Process the command line 4 ⟩;
    ⟨ Initialize everything 8 ⟩;
    ⟨ Input the clauses 9 ⟩;
    if (verbose & show_basics) ⟨ Report the successful completion of the input phase 21 ⟩;
    ⟨ Set up the main data structures 30 ⟩;
    imems = mems, mems = 0;
    ⟨ Solve the problem 39 ⟩;
    if (verbose & show_basics) fprintf(stderr,
        "Altogether␣%llu+%llu␣mems,␣%llu␣bytes,␣%llu␣nodes.\n", imems, mems, bytes, nodes);
  }
```

3.

```
#define show_basics 1 /* verbose code for basic stats */
#define show_choices 2 /* verbose code for backtrack logging */
#define show_details 4 /* verbose code for further commentary */
⟨ Global variables 3 ⟩ ≡
  int random_seed = 0; /* seed for the random words of gb_rand */
  int verbose = show_basics; /* level of verbosity */
  int hbits = 8; /* logarithm of the number of the hash lists */
  int buf_size = 1024; /* must exceed the length of the longest input line */
  ullng imems, mems; /* mem counts */
  ullng bytes; /* memory used by main data structures */
  ullng nodes; /* total number of branch nodes initiated */
  ullng thresh = 0; /* report when mems exceeds this, if delta ≠ 0 */
  ullng delta = 0; /* report every delta or so mems */
```

See also sections 7 and 26.

This code is used in section 2.

need SIZE(j). Instead, we can assume that the final literal of C_j is in location START($j-1$) $-$ 1, provided that we define START(0) appropriately.

The resulting procedure is almost unbelievably short and sweet. It's surely the simplest SAT solver that can claim to be efficient on problems of modest size:

Algorithm B (*Satisfiability by watching*). Given nonempty clauses $C_1 \wedge \cdots \wedge C_m$ on $n > 0$ Boolean variables $x_1 \ldots x_n$, represented as above, this algorithm finds a solution if and only if the clauses are satisfiable. It records its current progress in an array $m_1 \ldots m_n$ of "moves," whose significance was explained above.

B1. [Initialize.] Set $d \leftarrow 1$.

B2. [Rejoice or choose.] If $d > n$, terminate successfully. Otherwise set $m_d \leftarrow [W_{2d} = 0 \text{ or } W_{2d+1} \neq 0]$ and $l \leftarrow 2d + m_d$.

B3. [Remove $\bar{l}$ if possible.] For all j such that $\bar{l}$ is watched in C_j, watch another literal of C_j. But go to B5 if that can't be done. (See exercise 124.)

B4. [Advance.] Set $W_{\bar{l}} \leftarrow 0$, $d \leftarrow d + 1$, and return to B2.

B5. [Try again.] If $m_d < 2$, set $m_d \leftarrow 3 - m_d$, $l \leftarrow 2d + (m_d \,\&\, 1)$, and go to B3.

B6. [Backtrack.] Terminate unsuccessfully if $d = 1$ (the clauses are unsatisfiable). Otherwise set $d \leftarrow d - 1$ and go back to B5. ∎

Readers are strongly encouraged to work exercise 124, which spells out the low-level operations that are needed in step B3. Those operations accomplish essentially everything that Algorithm B needs to do.

SAT0W

SAT8

SAT9

Math/Physics

continued from page 1

Random formulas sampled from most distributions turned out to be easy to solve. Yet in the early 1990s, when considering formulas with a fixed ratio of clauses to variables, computer scientists noticed a curious phenomenon. When the ratio is small, formulas have many variables and few constraints; there are many satisfying assignments and it's easy for simple algorithms to find one of them. When the ratio is large, the variables tend to be overly constrained, and formulas almost certainly have no satisfying assignments. Remarkably, as the ratio, r, of clauses to variables grows, the transition from probably satisfiable to probably unsatisfiable is not gradual, but abrupt.

Researchers conjectured that there is a single threshold value of r, depending only on k, such that as the number of

unsatisfiable transit
threshold within a s
markably, the proof
help pinpoint the loc
old are now being ad
who are using them
sights rigorous.

Zeroing In on the

For the past two
scientists have been
value r_k that defines
almost surely satisfia
unsatisfiable. An upp
on r_k follows from th
ability of at least one
ment is bounded by
tends to 0 for $r > 2^k$
lower bounds were g
algorithms that can c
fying assignment up
These algorithms fai
per bound, howev
bounds of order $c2^k$

SAT10

SAT11

SAT13

SAT12

$$F(x_1, x_2, x_3) = (x_1 \vee \bar{x}_2) \wedge (x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3)$$

```
~ a simple example that's satisfiable
1 ~2
2 3
~1 ~3
~1 ~2 3
```

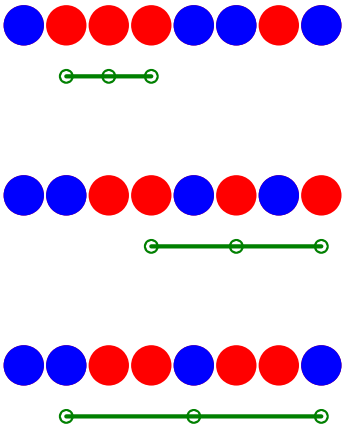
$$G(x_1, x_2, x_3) = F(x_1, x_2, x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3)$$

```
~ a simple example that's unsatisfiable
1 ~2
2 3
~1 ~3
~1 ~2 3
1 2 ~3
```

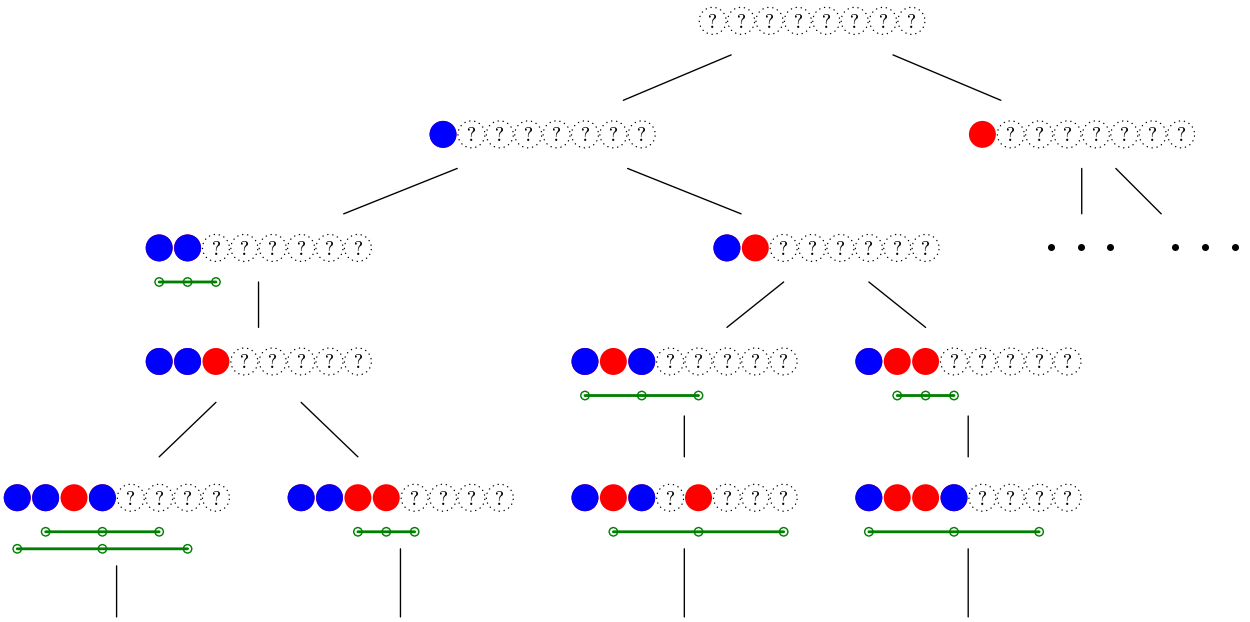
Van der Waerden's Problem

Place red and blue balls so that no three balls of the same color are equally spaced.

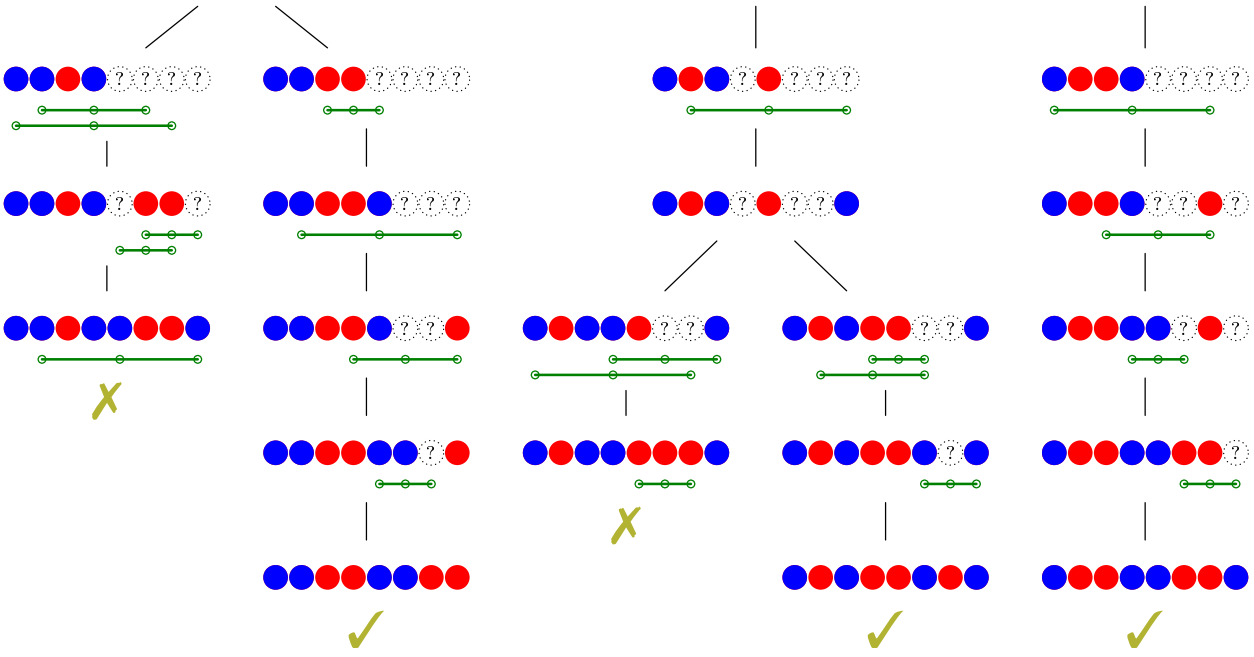
Examples of incorrect placements:



Try with eight balls:



Eight balls, continued:



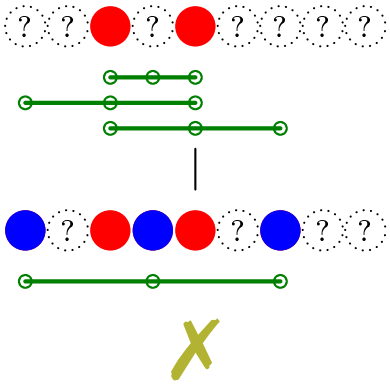
```
demo[1]> sat-waerden 3 3 8
~ sat-waerden 3 3 8
1 2 3
2 3 4
3 4 5
4 5 6
5 6 7
6 7 8
1 3 5
2 4 6
3 5 7
4 6 8
1 4 7
2 5 8
~1 ~2 ~3
~2 ~3 ~4
~3 ~4 ~5
 ...
~2 ~5 ~8
```

```
demo[2]> sat-waerden 3 3 8 | sat0
(8 variables, 24 clauses, 72 literals successfully read)
!SAT!
~1 2 ~3 4 5 ~6 7 ~8
Altogether 850+384 mems, 1712 bytes, 6 nodes.
demo[3]> sat-waerden 3 3 9 | sat0
(9 variables, 32 clauses, 96 literals successfully read)
~
UNSAT
Altogether 1111+9136 mems, 2200 bytes, 75 nodes.
demo[4]> sat-waerden 3 3 9 | sat0w
(9 variables, 32 clauses, 96 literals successfully read)
~
UNSAT
Altogether 222+3666 mems, 880 bytes, 79 nodes.
```

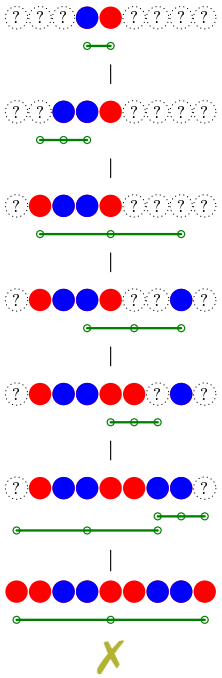
```
demo[5]> sat-waerden 3 3 9 | sat10
(9 variables, 32 clauses, 96 literals successfully read)
~
UNSAT
Altogether 252+5191 mems, 978 bytes, 7 nodes.
demo[6]> sat-waerden 3 3 9 | sat11
(9 variables, 32 clauses, 96 literals successfully read)
~
UNSAT
Altogether 1042+8987 mems, 7272 bytes, 3 nodes.
demo[7] sat-waerden 3 3 9 | sat13 s1001 l/tmp/learned
(9 variables, 32 clauses, 96 literals successfully read)
~
UNSAT
Altogether 667+2838 mems, 7021 bytes, 6 nodes.
6 learned clauses written to file ‘/tmp/learned’.
```

```
demo[8]> more /tmp/learned
 ~3 ~5
 4 ~5
 ~5
 3
 ~4 ~1
 ~1
```

```
demo[8]> more /tmp/learned
 ~3 ~5
 4 ~5
 ~5
 3
 ~4 ~1
 ~1
```



```
 ~3 ~5    ✓
 4 ~5
--More--(41%)
```



```
demo[8]> more /tmp/learned
 ~3 ~5    ✓
 4 ~5     ✓
 ~5
 3
 ~4 ~1
 ~1
```

. . . and so on: Each of the four remaining learned clauses can be deduced as a forced consequence of its predecessors. Finally, all six of the learned clauses force a contradiction. Thus the given clauses, which encode the nine-ball problem, are unsatisfiable.

```
demo[9]> sat-waerden 4 4 34 | sat13 >! /dev/null
(34 variables, 352 clauses, 1408 literals successfully read)
!SAT!
Altogether 4857+141856 mems, 21974 bytes, 132 nodes.
demo[10]> sat-waerden 4 4 35 | sat13 >! /dev/null
(35 variables, 374 clauses, 1496 literals successfully read)
UNSAT
Altogether 5137+448507 mems, 27663 bytes, 318 nodes.
```

```
demo[11]> sat-waerden 5 5 177 | sat13 >! /dev/null
(177 variables, 7656 clauses, 38280 literals successfully read)
!SAT!
Altogether 96579+74594130 mems, 619697 bytes, 11038 nodes.
demo[12]> sat-waerden 5 5 178 | sat13 l/tmp/learnt >! /dev/null
(178 variables, 7744 clauses, 38720 literals successfully read)
UNSAT
Altogether 97665+21013308385 mems, 3257686 bytes, 949207 nodes.
877397 learned clauses written to file ‘/tmp/learnt’.
75.324u 0.000s 1:15.37 99.9% 0+0k 0+0io 0pf+0w
```

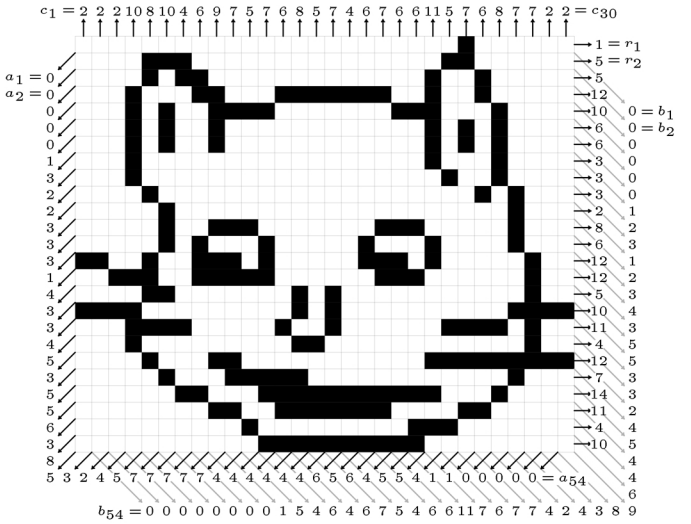
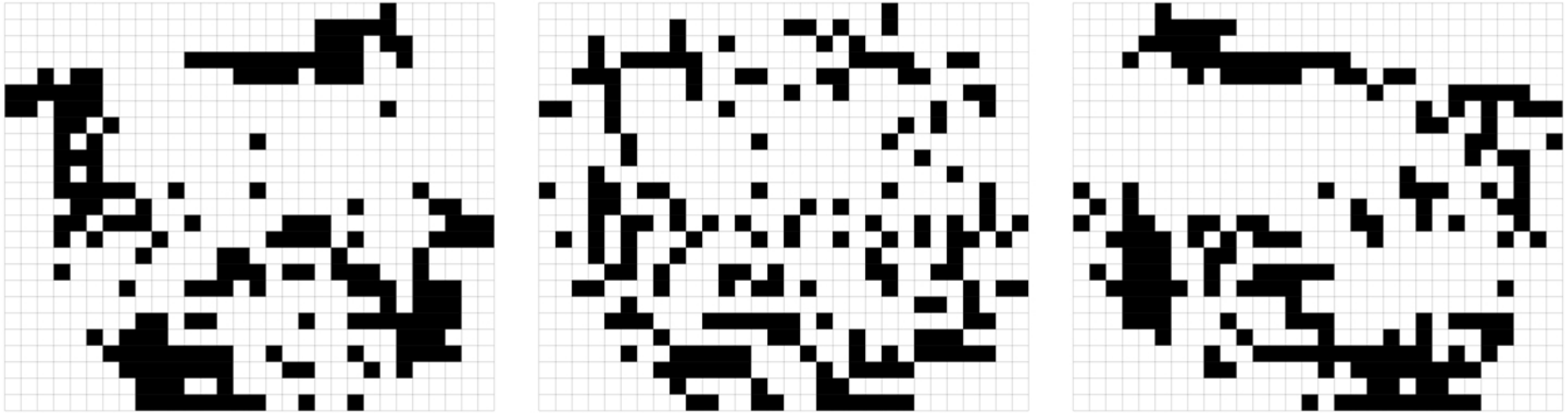


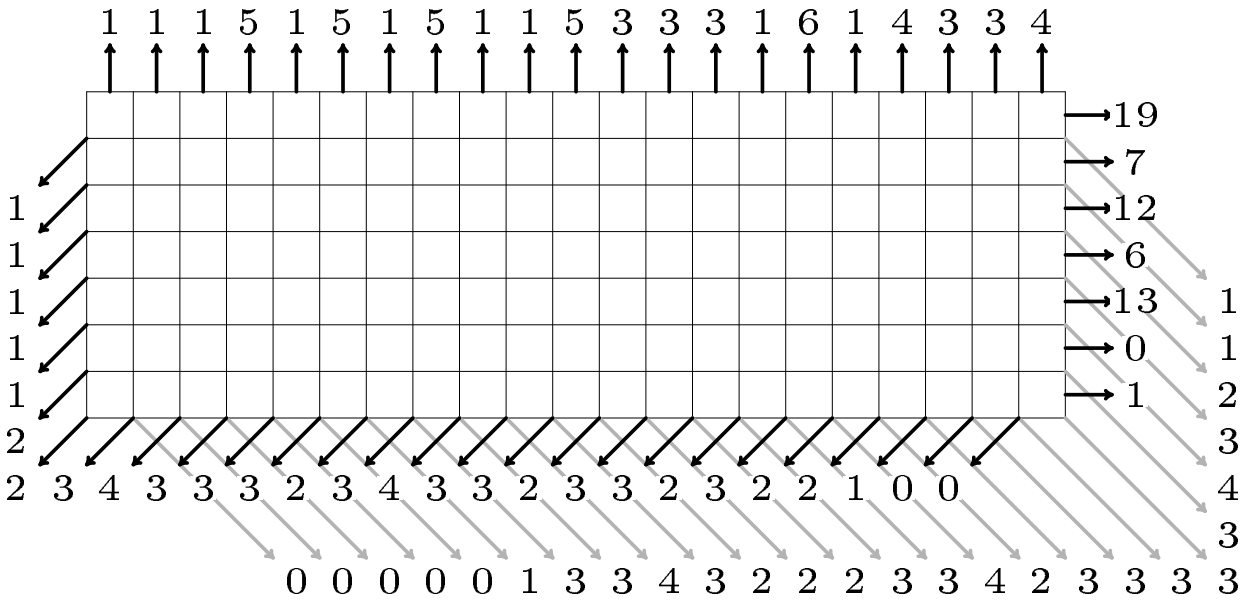
Fig. 36. An array of black and white pixels together with its row sums r_i, column sums c_j, and diagonal sums a_d, b_d.



(a) lexicographically first; (b) maximally different; (c) lexicographically last.

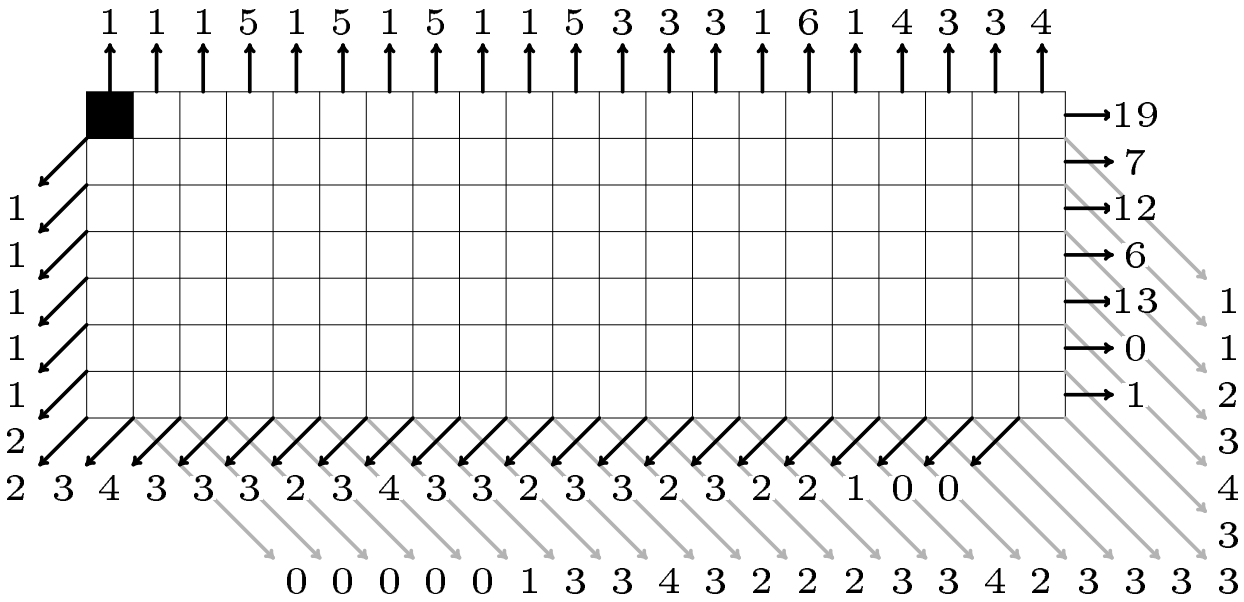
Fig. 37. Extreme solutions to the constraints of Fig. 36.

A Digital Tomography Puzzle



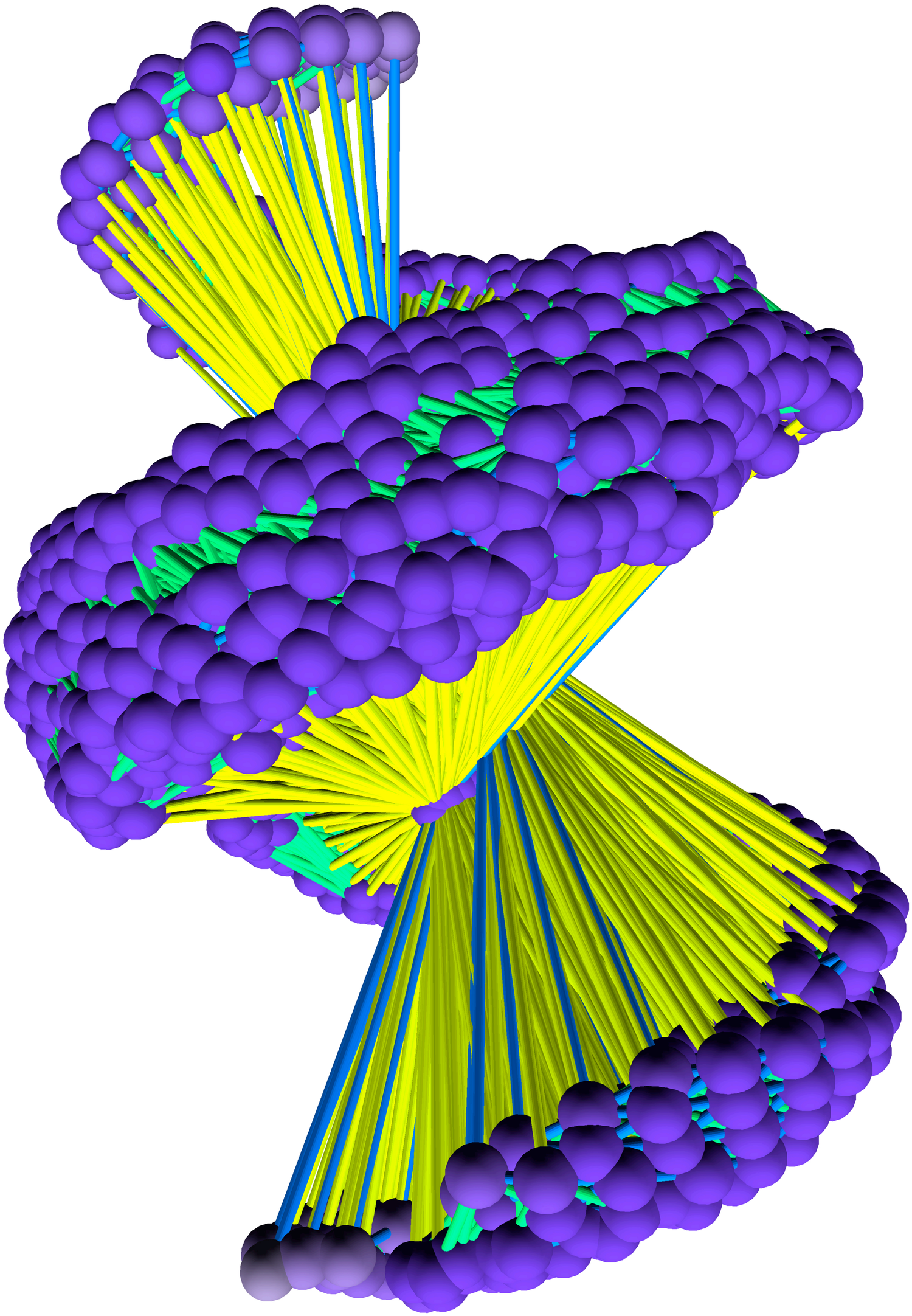
Each cell in this 7×21 array is either black or white.
The number of black cells in each row, column, and diagonal is shown.
Reconstruct the hidden black/white pattern.

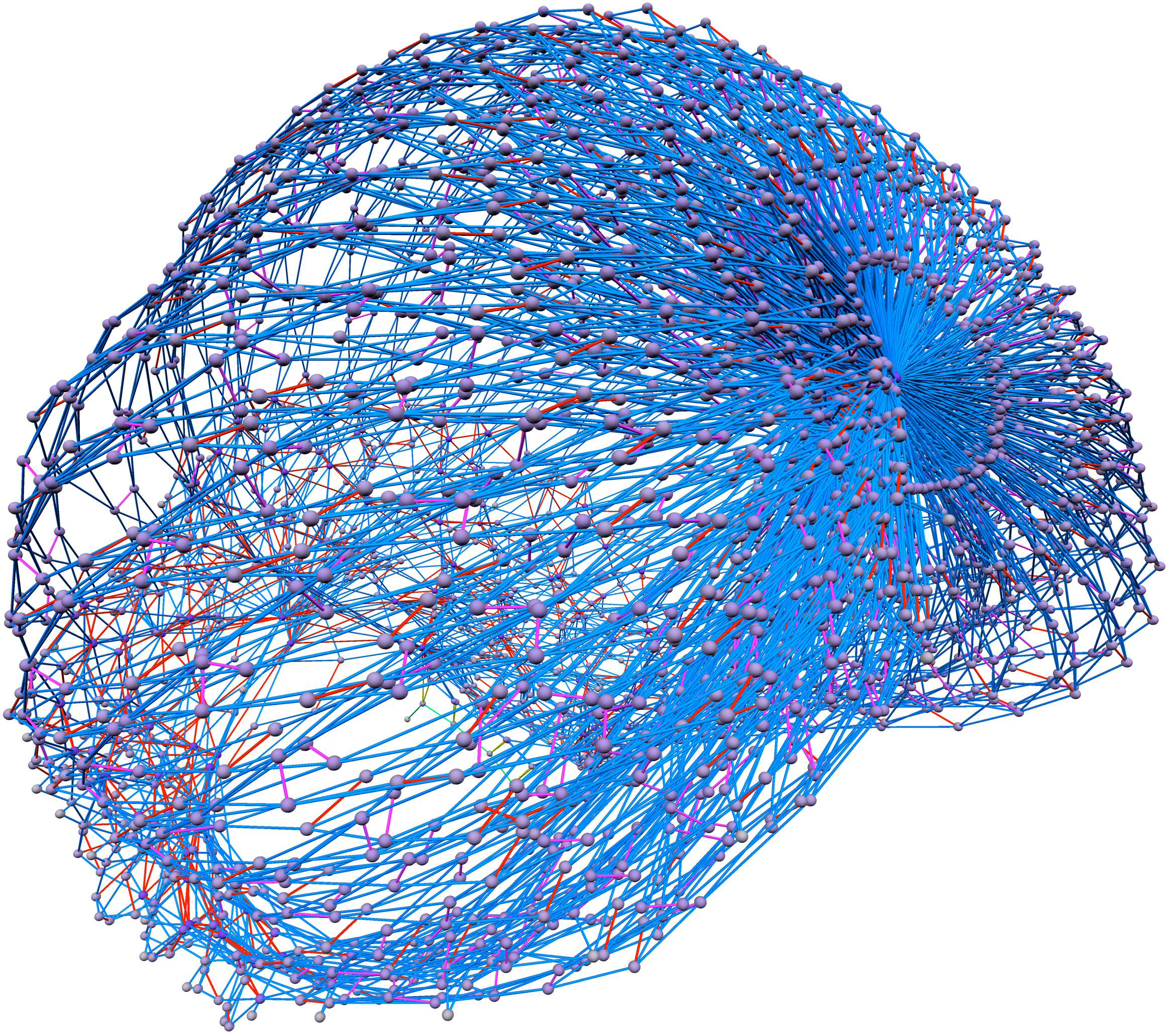
A Digital Tomography Puzzle — First step of solution

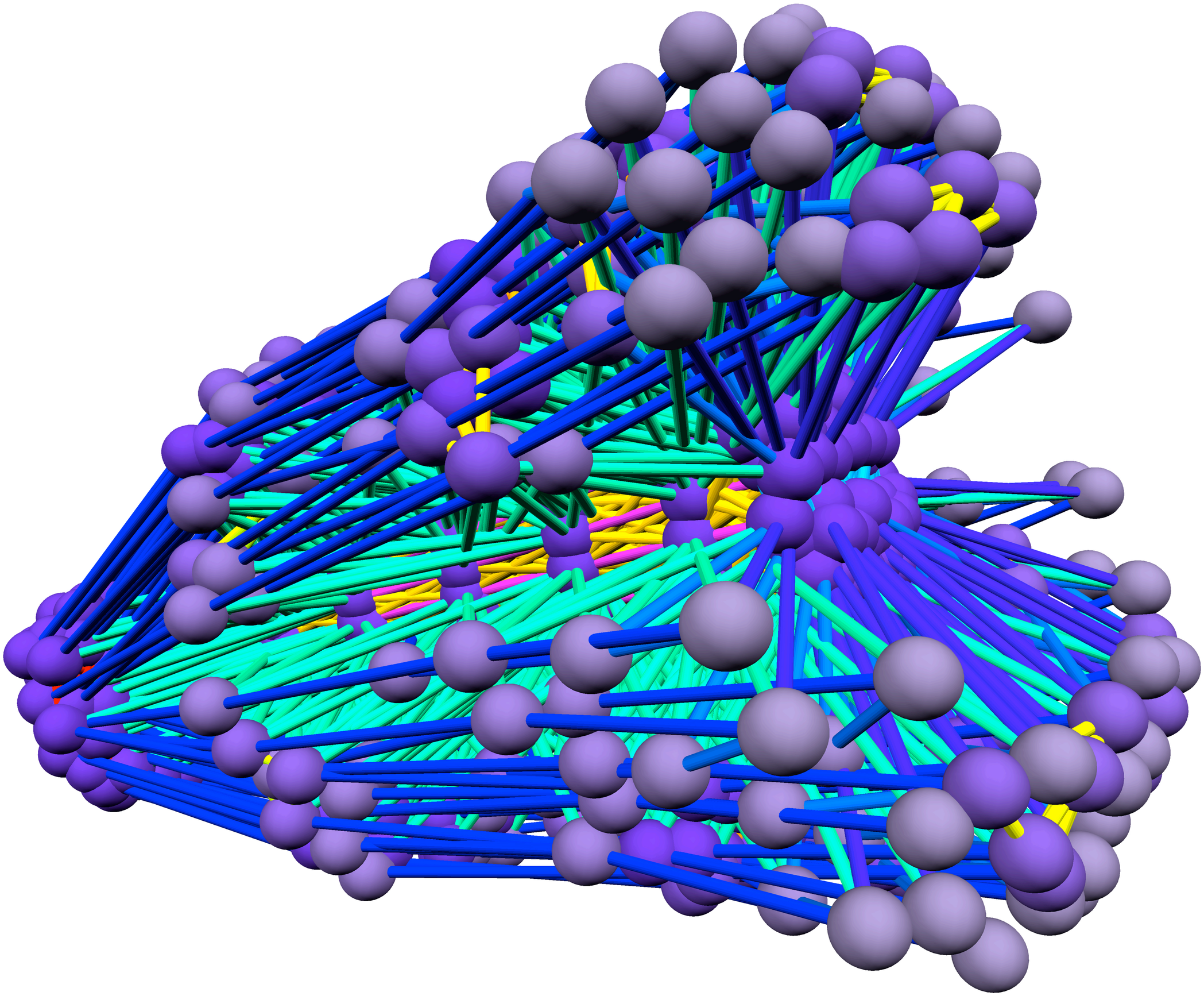


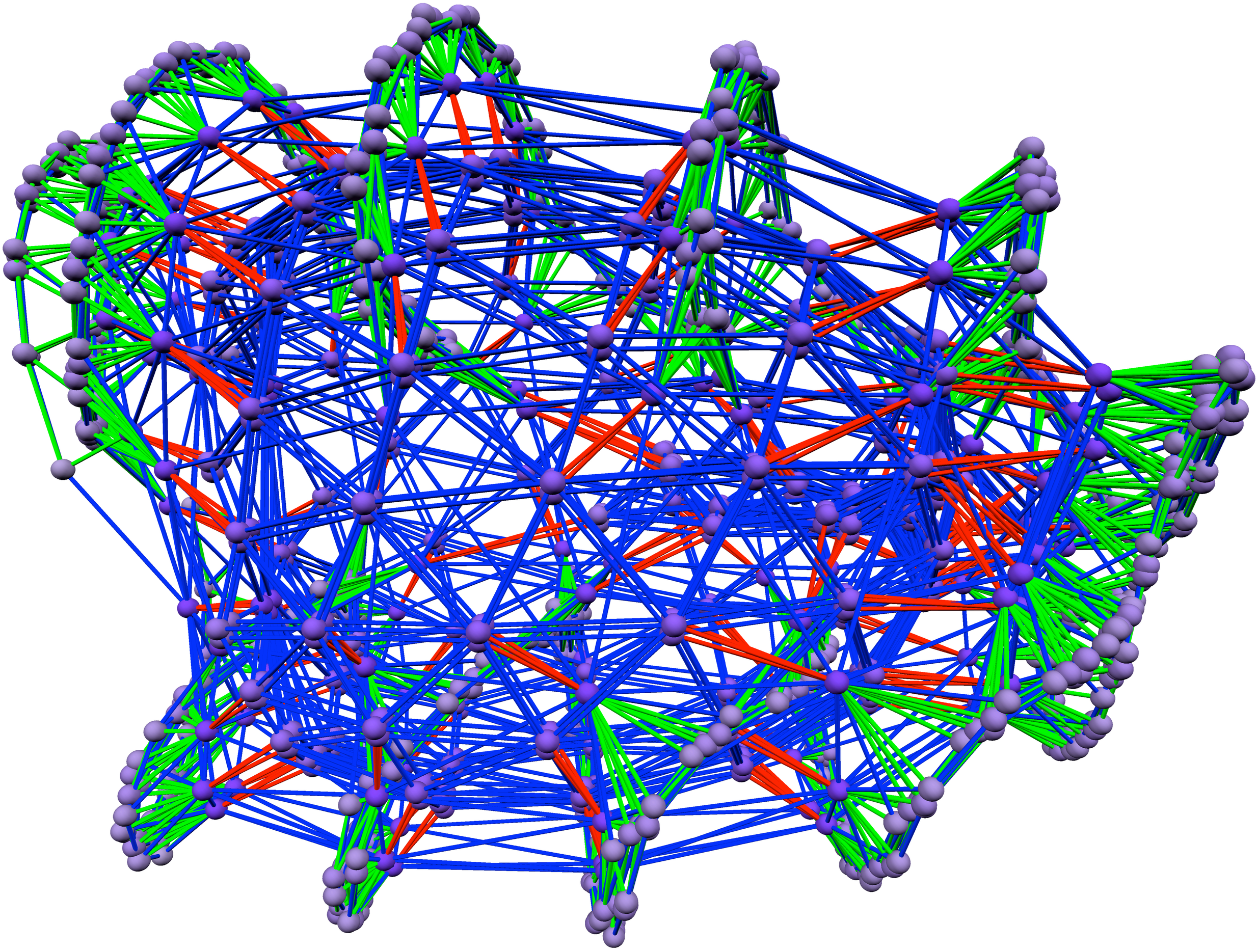
```
~ sat-tomography (7x21, 58)
1x21 1R1401
1x20 1R1401
1x20 1x21 1R1402
~1x20 ~1x21 ~1R1401
~1x20 ~1R1402
~1x21 ~1R1402
1x19 1R1301
1x18 1R1301
1x18 1x19 1R1302
~1x18 ~1x19 ~1R1301
~1x18 ~1R1302
~1x19 ~1R1302
1x17 1R1201
1x16 1R1201
1x16 1x17 1R1202
~1x16 ~1x17 ~1R1201
~1x16 ~1R1202
~1x17 ~1R1202
 ...
```

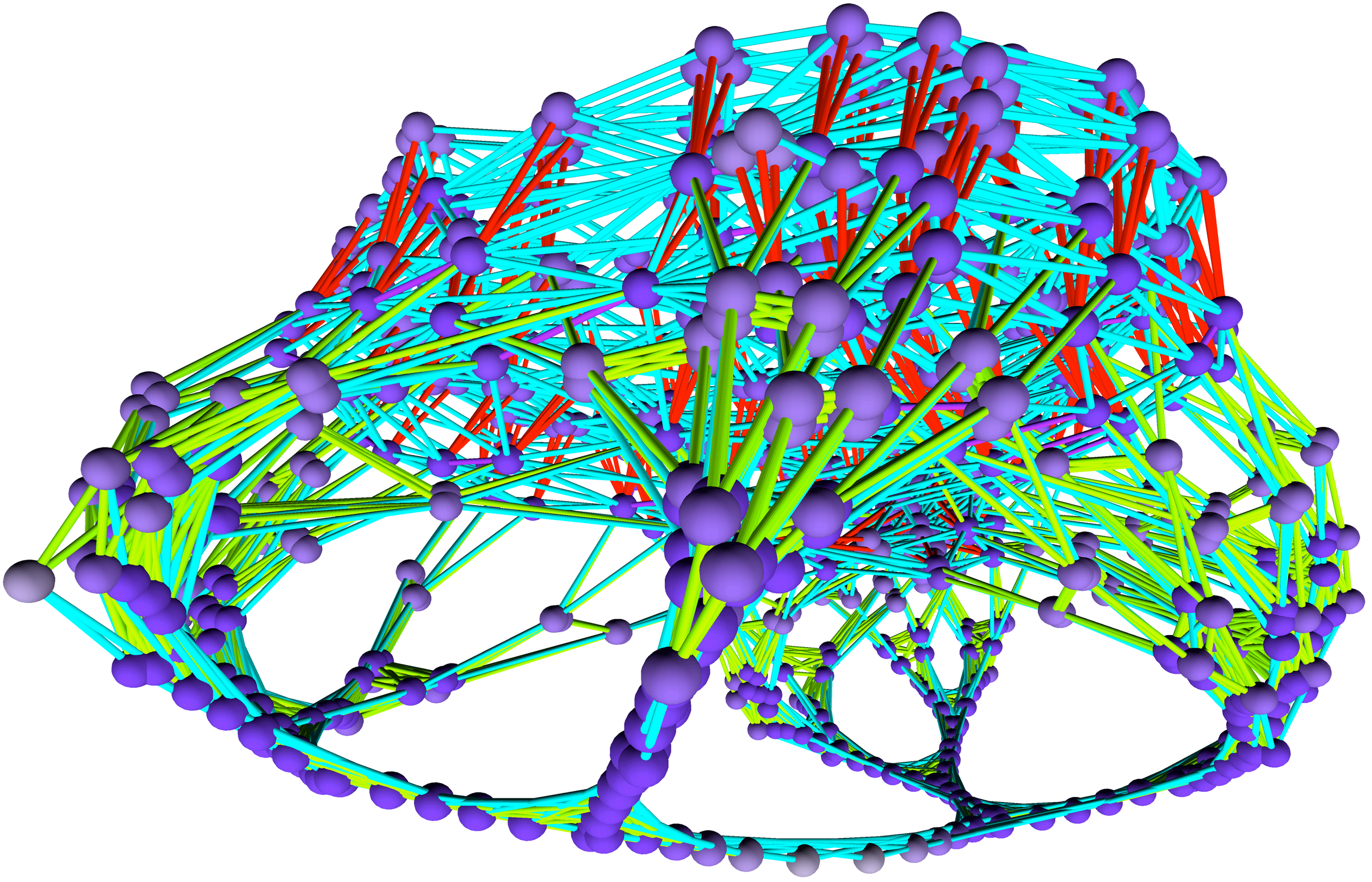
```
demo[13]> sat-tomography-prep 7 21 < puzzle.data | sat-tomography | sat13 >! /dev/null
OK, I've input an image with 7 rows and 21 columns.
Input for 7 rows and 21 columns successfully read (total 58)
(1039 variables, 3926 clauses, 9362 literals successfully read)
!SAT!
Altogether 77340+30907 mems, 201223 bytes, 0 nodes.
```

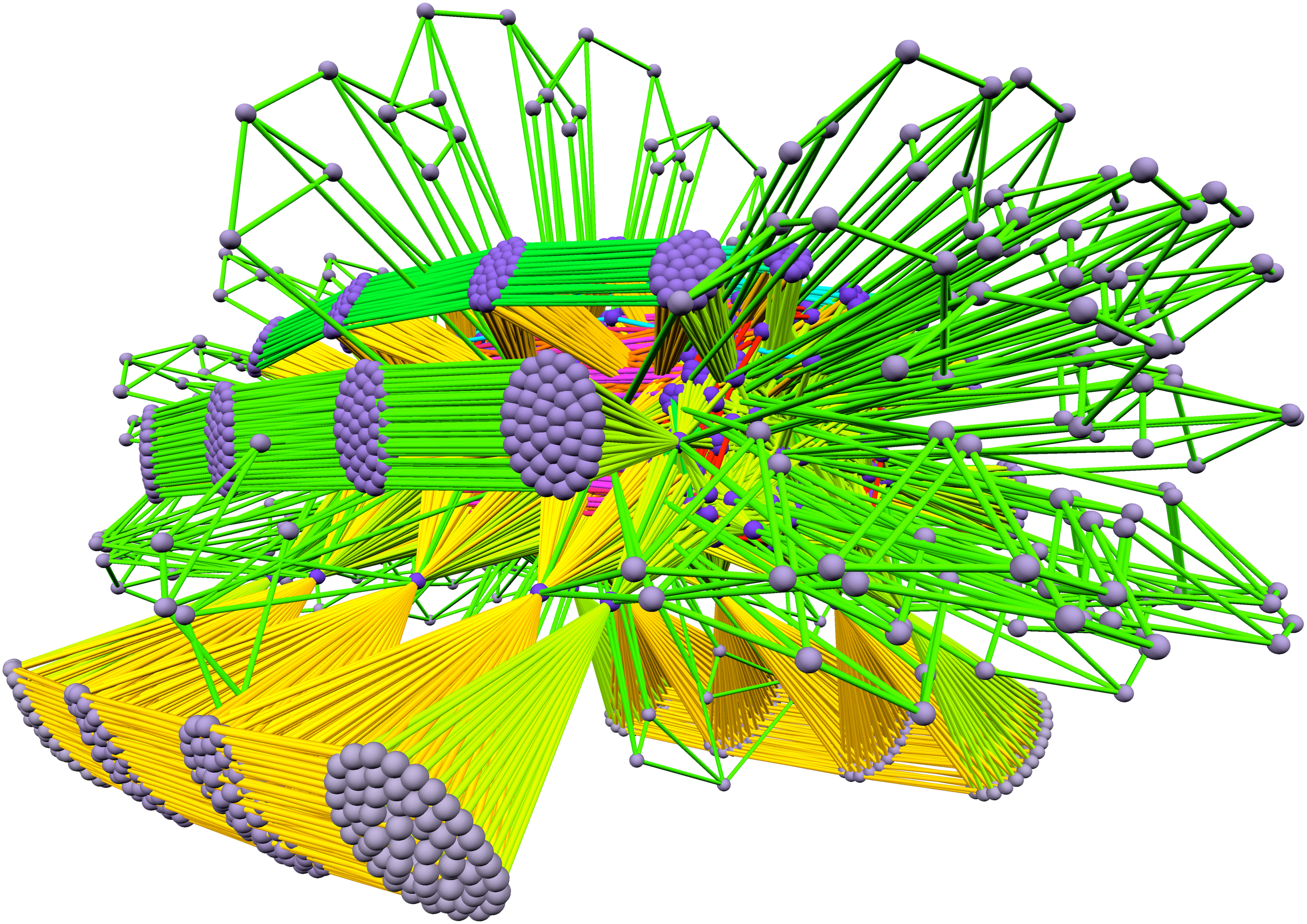


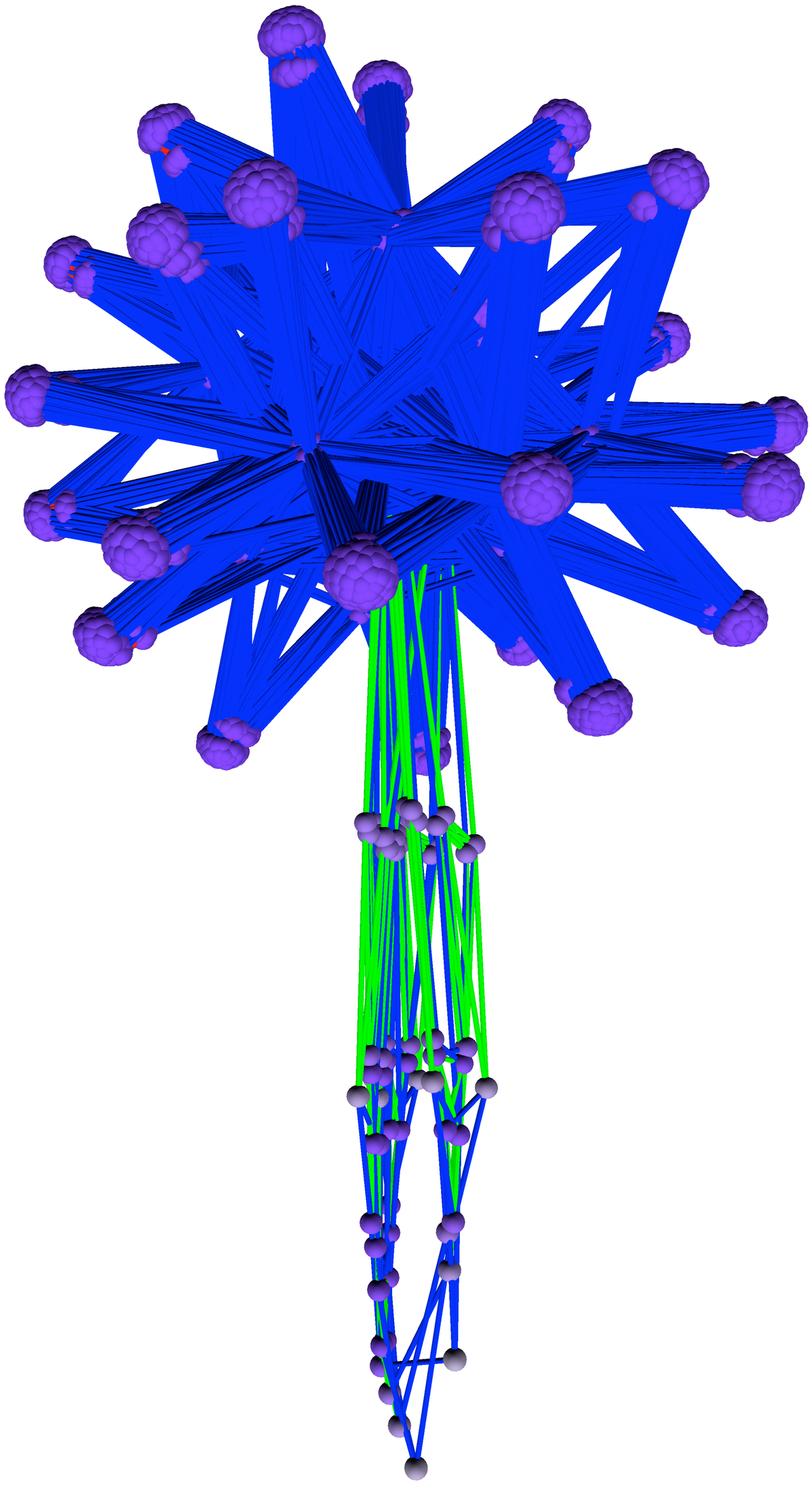


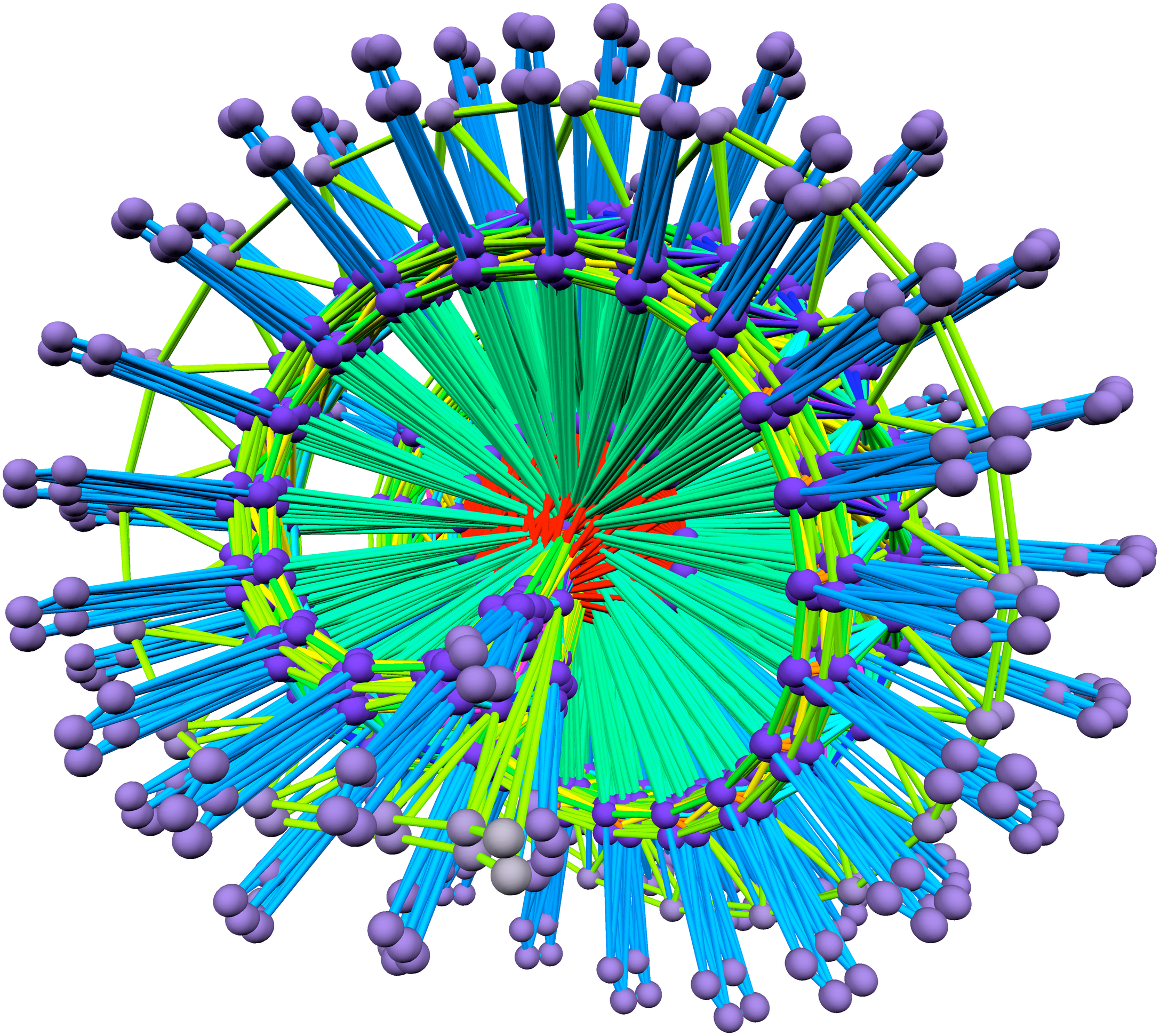






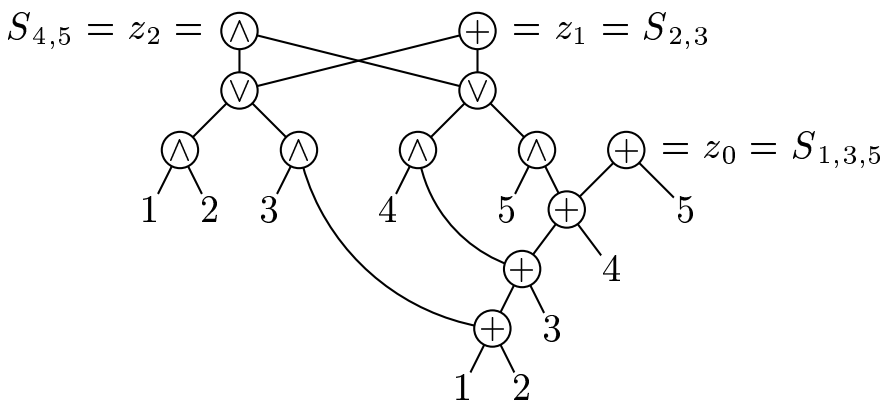




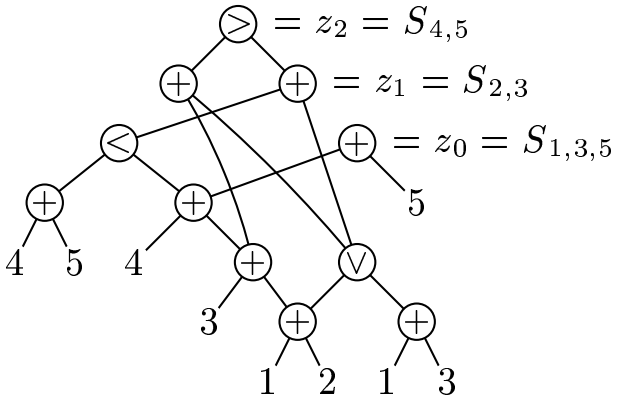


Synthesis of a circuit $(x_1 + x_2 + x_3 + x_4 + x_5 = 4z_2 + 2z_1 + z_0)$

The old way:



The new way:
(found by solving 76321
clauses in 957 variables)



The Pythagorean triples problem: *Choose either* 1 *or* 1, 2 *or* 2, ..., N *or* N, *without having* $a^2 + b^2 = c^2$ *or* $a^2 + b^2 = c^2$.

Graham, 1984: Is it possible for $N = \infty$? ($100 reward!)

Cooper, Poirel, 2008: Yes for $N = 1344$.

Kay, 2000: Yes for $N = 1514$.

Myers, 2015: Yes for $N = 6500$.

Cooper, Overstreet, 2015: Yes for $N = 7664$.

Heule, Kullmann, Marek, 2016: Yes for $N = 7824$.

Heule, Kullmann, Marek, 2016: No for $N = 7825$!

Solving and Verifying the boolean Pythagorean Triples problem via Cube-and-Conquer

Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek

The University of Texas at Austin, Swansea University, and University of Kentucky

Abstract. The *boolean Pythagorean Triples problem* has been a long-standing open problem in Ramsey Theory: Can the set $\mathbb{N} = \{1, 2, \dots\}$ of natural numbers be divided into two parts, such that no part contains a triple (a, b, c) with $a^2 + b^2 = c^2$? A prize for the solution was offered by Ronald Graham over two decades ago. We solve this problem, proving in fact the impossibility, by using the *Cube-and-Conquer* paradigm, a hybrid SAT method for hard problems, employing both look-ahead and CDCL solvers. An important role is played by dedicated look-ahead heuristics, which indeed allowed to solve the problem on a cluster with 800 cores in about 2 days. Due to the general interest in this mathematical problem, our result requires a formal proof. Exploiting recent progress in unsatisfiability proofs of SAT solvers, we produced and verified a proof in the DRAT format, which is almost 200 terabytes in size. From this we extracted and made available a compressed certificate of 68 gigabytes, that allows anyone to reconstruct the DRAT proof for checking.

1 Introduction

Propositional satisfiability (SAT, for short) is a formalism that allows for representation of all finite-domain constraint satisfaction problems. Consequently, all decision problems in the class NP, as well as all search problems in the class FNP [9,29,35,19], can be polynomially reduced to SAT. Due to great progress with *SAT solvers*, many practically important problems are solved using such

```
~ sat-pyth 7825
3 4 5
~3 ~4 ~5
5 12 13
~5 ~12 ~13
6 8 10
~6 ~8 ~10
7 24 25
~7 ~24 ~25
8 15 17
~8 ~15 ~17
9 12 15
~9 ~12 ~15
9 40 41
~9 ~40 ~41
10 24 26
~10 ~24 ~26
 ...
5474 5520 7774
~5474 ~5520 ~7774
```

```
demo[14]> sat-pyth 6500 | sat13 d10000000000 f.001 h10 s3142 >
! /dev/null
(5370 variables, 15348 clauses, 46044 literals successfully re
ad)
  after 100000053109 mems:  z=0 d=45.7 t=515.1 m=47084.3 p=302.
9 m/p=155.4 r=34.3 L=37.9 l=35.4 g=30.7 s=0.02 a=0.13
!SAT!
Altogether 293197+145227517488 mems, 11971290 bytes, 4527737 n
odes.
749.386u 0.024s 12:30.48 99.8% 0+0k 0+0io 0pf+0w
```

```
demo[14]> sat-pyth 6500 | sat13 d10000000000 f.001 h10 s3142 >
! /dev/null
(5370 variables, 15348 clauses, 46044 literals successfully re
ad)
  after 100000053109 mems:  z=0 d=45.7 t=515.1 m=47084.3 p=302.
9 m/p=155.4 r=34.3 L=37.9 l=35.4 g=30.7 s=0.02 a=0.13
!SAT!
Altogether 293197+145227517488 mems, 11971290 bytes, 4527737 n
odes.
749.386u 0.024s 12:30.48 99.8% 0+0k 0+0io 0pf+0w
```

(The calculations by Heule, Kullman, and Marek for $N = 7825$, using the `glucose` 3.0 solver by Gilles Audemard and Laurent Simon, took about 35,000 hours of CPU time, after dividing the task into about a million independent jobs to be run in parallel. But the computation was completed in two days, because they did it in Texas — on a computer cluster called Stampede.)