Noname manuscript No.
(will be inserted by the editor)

Binary pattern tile set synthesis is NP-hard

Lila Kari · Steffen Kopecki · Pierre-Étienne Meunier · Matthew J. Patitz · Shinnosuke Seki

Received: date / Accepted: date

Abstract We solve an open problem, stated in 2008, about the feasibility of designing efficient algorithmic self-assembling systems which produce 2-dimensional colored patterns. More precisely, we show that the problem of finding the smallest tile assembly system which rectilinearly self-assembles an input pattern with 2 colors (i.e., 2-PATS) is **NP**-hard. Of both theoretical and practical significance, the more general k-PATS problem has been studied in a series of papers which have shown k-PATS to be **NP**-hard for $k = 60$, $k = 29$, and then $k = 11$. In this paper, we prove the fundamental conjecture that 2-PATS is **NP**-hard, concluding this line of study.

While most of our proof relies on standard mathematical proof techniques, one crucial lemma makes use of a computer-assisted proof, which is a relatively novel but increasingly utilized paradigm for deriving proofs for complex

This is a full version of [20]

This work is supported in part by the NSERC Discovery Grant R2824A01 and UWO Faculty of Science grant to L. K., NSF Grant CCF-1219274, NSF Grants CCF-1117672 and CCF-1422152, Academy of Finland, Postdoctoral Researcher Grant 13266670/T30606 to S. S., JST Program to Disseminate Tenure Tracking System, MEXT, Japan 6F36 to S. S., and JSPS Grant-in-Aid for Research Activity Start-up to S. S.

Lila Kari and Steffen Kopecki
Department of Computer Science, University of Western Ontario, London ON N6A 1Z8, Canada. E-mail: {lila,steffen}@csd.uwo.ca

Pierre-Étienne Meunier
Department of Computer Science, Aalto University, P.O.Box 15400, FI-00076, Aalto, Finland. E-mail: pierre-etienne.meunier@aalto.fi

Matthew J. Patitz
Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR, USA. E-mail: mpatitz@self-assembly.net

Shinnosuke Seki
Department of Communication Engineering and Informatics, University of Electro-Communications, 1-5-1, Chofugaoka, Chofu, Tokyo, 1828585, Japan. E-mail: s.seki@uec.ac.jp

mathematical problems. This tool is especially powerful for attacking combinatorial problems, as exemplified by the proof for the four color theorem and the recent important advance on the Erdős discrepancy problem using computer programs. In this paper, these techniques will be brought to a new order of magnitude, computational tasks corresponding to one CPU-year. We massively parallelize our program, and provide a full proof of its correctness. Its source code is freely available online.

Keywords Algorithmic DNA self-assembly · Pattern assembly · **NP**-hardness · Computer-assisted proof · Massively-parallelized program

1 Introduction

The traditional way for humankind to modify the physical world has been via a top-down process of crafting things with tools, in which matter is directly manipulated and shaped by those tools. In this work, we are interested in another crafting paradigm called *self-assembly*, a model of building structures from the bottom up. Via self-assembly, it is possible to design molecular systems so that their components autonomously combine to form structures with nanoscale, even atomic, precision. At this scale, tools are no longer the easiest way to build things, and *programming* the assembly of matter becomes at the same time easier, cheaper, and more powerful.

Using this paradigm, researchers have already built a number of things, such as regular arrays [46], fractal structures [12, 35], logic circuits [30, 37], maps [34, 44], DNA tweezers [49], neural networks [31], and molecular robots [24], just to name a few. Such examples demonstrate that self-assembly can be used to manufacture specialized geometrical, mechanical, and computational objects at the nanoscale. Potential future applications of nanoscale self-assembly include the production of new materials with specifically tailored properties (electronic, photonic, etc.) and medical technologies which are capable of diagnosing and even treating diseases in vivo, at the cellular level. Furthermore, studying the processes occurring in self-assembling systems yields precious insights about what is physically, even theoretically, possible in these molecular systems. Questions such as "what is the smallest program capable of performing a given task?" arise naturally in these systems, either from experimental applications, or from more fundamental research on the capabilities of natural systems.

The *abstract Tile Assembly Model* (aTAM) was introduced by Winfree [45] to study the possibilities brought by molecular components built by Seeman [38] using DNA. This model is essentially an asynchronous nondeterministic cellular automaton, and can also be seen as a dynamical variant of Wang tiling [43]. In the aTAM, the basic components are translatable but unrotatable square *tiles* whose sides are labeled with *glues*, each with an integer *strength*. Growth proceeds from a *seed assembly*, one tile at a time, and at each time step a tile can attach to an existing assembly if the sum of the strengths of the glues on its sides, whose types match the existing assembly, is equal to at least a parameter of the model called the *temperature*. Despite its deliberate simplification, the aTAM is a computationally expressive model [23, 29, 45] capable of Turing universal computation. Recently, it has even been shown to be intrinsically universal [8, 9, 10, 11, 27, 47].

1.1 **NP**-hardness of pattern tile set synthesis

The problem we study in this paper is the optimization of the design of tile assembly systems in the aTAM which self-assemble to form colored input patterns. DNA tiles can be equipped with proteins [48] and nanoparticles such as gold (`Au`) [50]. Assemblies of normal tiles as well as tiles thus modified can be considered a *colored pattern*, as a periodic placement of `Au` nanoparticles on a 2D nanogrid [50] can be considered a 2-colored (i.e., binary) rectangular pattern on which the two colors specify the presence/absence of an `Au` nanoparticle at the position. Various designs of pattern assemblers have been proposed theoretically and experimentally; see for example [4, 6, 35, 50]. In general, k-PATS for $k \geq 2$ is the task, given a placement of k different kinds of nanoparticles, represented in the model as a k-colored rectangular pattern, to design an optimally small tileset and an L-shaped seed that self-assembles the pattern; see Fig. 1 for an example. Essentially, each type of tile is assigned a "color", and the goal is to design a system consisting of the minimal number of tile types such that they deterministically self-assemble to form a rectangular assembly in which each tile is assigned the same color as the corresponding location in the pattern. This problem was introduced in [25], and has since then been extensively studied [7, 15, 18, 19, 39]. The interest is both theoretical, to determine the computational complexity of designing efficient tile assembly systems, and practical, as the goal of self-assembling patterned substrates onto which a potentially wide variety of molecular components could be attached is a major experimental goal. In [39] Seki proved for the first time the **NP**-hardness of 60-PATS (i.e., the input pattern is allowed to have 60 colors) and the result has since been strengthened to that of 29-PATS [18], and further to 11-PATS [19]. Additionally, a variant of k-PATS, where the number of tile types of certain colors is restricted, has been proven to be **NP**-hard for 3 colors [21].

The foundational conjecture has been that for $k = 2$, that is, 2-PATS, the problem is also **NP**-hard as stated in 2008[1]. This open problem in the field of DNA self-assembly is known as *binary pattern tile set synthesis* (2-PATS) problem [25, 39]. Our main result confirms this conjecture, which is thus the terminus of this line of research and a fundamental result in algorithmic self-assembly. We state the main result of this paper here, although some terms may not be formally defined yet:

Theorem 1 *The* 2-PATS *optimization problem of finding, given a* 2 *colored rectangular pattern* P*, the minimal colored tileset (together with an L-shape*

[1] This problem was claimed to be **NP**-hard in a subsequent paper by the authors of [25] but what they proved was the **NP**-hardness of a different problem (see [40]).

seed) that produces a single terminal assembly where the color arrangement is exactly the same as in P, is **NP***-hard.*

The main idea of our proof is similar to the strategies adopted by [18, 19, 39]. We embed the computation of a verifier of solutions for an **NP**-complete problem (in our case, a variant of SAT, which we call M-SAT) in an assembly, which is relatively straightforward in Winfree's aTAM. One can indeed engineer a tile assembly system (TAS) in this model, with colored tiles, implementing a verifier of solutions of the variant of SAT, in which a formula F and a variable assignment $\phi \in \{0,1\}^n$ are encoded in the seed assembly, and a tile of a special color appears in a certain position if and only if $F(\phi) = 1$. In our actual proof, reported in Sect. 3, we design a set T of 13 tile types and a reduction of a given instance ϕ of M-SAT to a rectangular pattern P_F such that

Property 1. A TAS using tile types in T self-assembles P_F iff F is satisfiable.

Property 2. Any TAS of at most 13 tile types that self-assembles P_F is isomorphic to T.

Therefore, F is solvable if and only if P_F can be self-assembled using at most 13 tile types. In previous works [18, 19, 39], significant portions of the proofs were dedicated to ensure their analog of Property 2, and many colors were "wasted" to make the property "manually" checkable. For reference, 33 out of 60 colors just served this purpose for the proof of **NP**-hardness of 60-PATS [39] and 2 out of 11 did that for 11-PATS [19]. Cutting this "waste" causes a combinatorial explosion of cases to test and motivates us to use a computer program to do the verification instead. Apart from the verification of Property 2 (in Lemma 1), the rest of our proof can be verified as done in traditional mathematical proofs; our proof is in Sect. 3.

The verification of Property 2 is done by an algorithm which, given a pattern and an integer n, searches for all possible sets of n tile types that self-assemble the pattern. We provide two parallelized implementations of the algorithm: a fast, unproven C++ version, and a slower, but formally proven OCaml implementation. A high-level explanation of the algorithm and the two implementations is given in Sect. 4 and both implementations are freely available online[2]. Both versions were implemented independently and neither is the conversion of the code of the other implementation. The full statistics of the runs are available on demand, and summarized by the Parry user interface: `http://pats.lif.univ-mrs.fr`.

1.2 Computer-assisted proofs

In one of its parts, our proof of the 2-PATS conjecture requires the solution of a massive combinatorial problem, meaning that one of the lemmas upon which

[2] `http://self-assembly.net/wiki/index.php?title=2PATS-tileset-search` (C++ version) and `http://self-assembly.net/wiki/index.php?title=2PATS-search-ocaml` (OCaml version)

it relies needs a massive exploration of more than $6 \cdot 10^{13}$ cases via a computer program. While this is not a traditional component of mathematical proofs, and may not provide the same level of insight into *why* something is true that a standard proof may, modern hardware and software have now given us the tools to attack combinatorially formidable problems whose proofs, if not augmented by computer programs, would often be impossible or as lacking in their ability to elucidate the reasons for their truth due to explosive case analyses as verification by brute force analysis of a computer program. Indeed, computer science has at the same time introduced combinatorial arguments indicating that most theorems do not have simple proofs, and possible ways to produce *certain facts* anyway, by heavy algorithmic processes. Moreover, the "natural proofs" line of research [1, 5, 32, 36] suggests that understanding "why" complexity classes are separated may be out of reach, and that therefore, the study of these kinds of proofs, and methods to ensure their correctness, are a fundamental direction in computer science today. Asserting the correctness of biological and chemical programs is also an important problem, where "*why*" questions are really not as important as the "*whether*" ones, for instance for therapeutic applications. Computationally intensive proofs are therefore likely to become common in these areas of science.

Historically, Appel and Haken [2, 3] were the first to prove a result — the four color theorem — with this kind of method, in 1976. This proof was later simplified in [33]. Since then, important problems in various fields have been solved (fully or partially) with the assistance of computers: the discovery of Mersenne primes [42], the 17-point case of the happy ending problem [41], the **NP**-hardness of minimum-weight triangulation [28], a special case of Erdős' discrepancy conjecture [22], the ternary Goldbach conjecture [17], and Kepler's conjecture [16, 26], among others. Over the years, exhaustive exploration and massively parallel programs have also been commonly used in physics, or in combinatorial problems such as solving the Rubik's cube. However, none of these programs was proven formally, and confidence in the validity of these results thus relies on our trust in the programmers.

The first rigorous proof of a massive software exploration was for the four colors theorem, recently done in the Coq proof assistant by Gonthier et al. [14]. The order of magnitude of their proof is close to the limits of Coq, and is not comparable with our result, which needs a massively parallel exploration requiring about one CPU-year on very modern, high-end machines (as a sum total over several hundred distributed cores) to complete and verify the correctness of the lemma.

Unless the implementation of assistant computer programs is straightforward, we need to make a strategic plan to tackle a problem so meticulously that human beings can verify the computer programs employed and their underlying algorithms rigorously. Such efforts may lead us to further theoretical developments and deeper insights into the problem, as a new proof of the four color theorem by Robertson et al. benefited from the improved time complexity of map-verification algorithms and the reduced number, 633, of candidates to be checked [33].

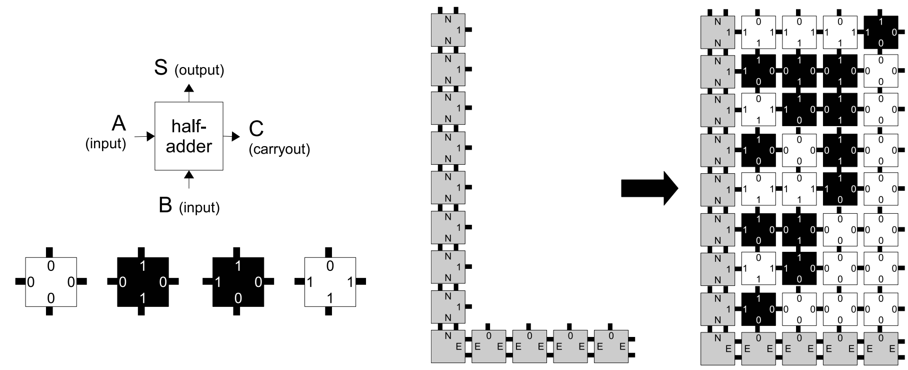


Fig. 1 (Left) Four tile types implement the half-adder with two inputs A, B from the west and south, the output S to the north, and the carryout C to the east; (Right) Copies of the half-adder tiles turn the L-shape seed into the binary counter pattern

A large parallel cluster was hence employed, which poses a number of new challenges. Indeed, in a sequential program, we often implicitly use the fact that function calls return the output of their computations, which becomes more complicated when using multiple computers: without using unrealistic hypotheses on the correction of the network and of operating systems, return values could potentially be lost, duplicated or corrupted. Since our program ran for a long time, we cannot make such strong hypotheses, which is why we need to assert the authenticity of messages received by the server by using cryptographic signatures.

Another feature of our proof is the use of a *functional programming language*, OCaml. The main feature of this language is the conciseness of the code, and the proximity of its syntax to mathematical proofs. In Section 5, we present a full proof of our programs, for the sake of completeness. This is not to be confused with an *explanation* of the code, which is given in Section 4: it is rather a rigorous argument to show that the statement of Lemma 1 holds. The whole framework for carrying out the programmatic part of our proof is reusable for the same kind of tasks in the future.

2 Preliminaries

Let $\mathbb{N}$ be the set of nonnegative integers, and for a positive integer $n \in \mathbb{N}$, let $[n] = \{0, 1, 2, \ldots, n-1\}$. For $k \geq 1$, a *k-colored pattern* is a partial function from $\mathbb{N}^2$ to the set of (color) indices $[k]$, and a *k-colored rectangular pattern* (of width w and height h) is a pattern whose domain is $[w] \times [h]$.

Let Σ be a glue alphabet. A *(colored) tile type* t is a tuple $(g_{\mathtt{N}}, g_{\mathtt{W}}, g_{\mathtt{S}}, g_{\mathtt{E}}, c)$, where $g_{\mathtt{N}}, g_{\mathtt{W}}, g_{\mathtt{S}}, g_{\mathtt{E}} \in \Sigma$ represent the respective north, west, south, and east glue of t, and $c \in \mathbb{N}$ is a color (index) of t. For instance, the right black tile type in Fig. 1 (Left) is (1, 1, 0, 0, black). We refer to $g_{\mathtt{N}}, g_{\mathtt{W}}, g_{\mathtt{S}}, g_{\mathtt{E}}$ as $t(\mathtt{N}), t(\mathtt{W}), t(\mathtt{S}), t(\mathtt{E})$, respectively, and by $c(t)$ we denote the color of t. For a set T of tile types, an *assembly* α over T is a partial function from $\mathbb{N}^2$ to T. When algorithms and computer programs will be explained in Sect. 4, it is convenient for the tile types in T to be indexed as $t_0, t_1, \ldots, t_{\ell-1}$ and consider the assembly rather as a partial function from $\mathbb{N}^2$ to $[\ell]$. Its pattern, denoted by $P(\alpha)$, is such that $\mathrm{dom}(P(\alpha)) = \mathrm{dom}(\alpha)$ and $P(\alpha)(x, y) = c(\alpha(x, y))$ for any $(x, y) \in \mathrm{dom}(\alpha)$. Given another assembly β, we say α is a *subassembly* of β if $\mathrm{dom}(\alpha) \subseteq \mathrm{dom}(\beta)$ and, for any $(x, y) \in \mathrm{dom}(\alpha)$, $\beta(x, y) = \alpha(x, y)$.

A *rectilinear tile assembly system* (RTAS) is a pair $\mathcal{T} = (T, \sigma_L)$ of a set T of tile types and an L-shape seed σ_L. The seed σ_L is an assembly over another set of tile types disjoint from T such that $\mathrm{dom}(\sigma)_L = \{(-1, -1)\} \cup ([w] \times \{-1\}) \cup (\{-1\} \times [h])$ for some $w, h \in \mathbb{N}$. The *vertical arm (of the seed)* consists of those tiles of the seed lying on the column with x-coordinate -1; the other tiles of the seed, lying on the row with y-coordinate -1, make up the *horizontal arm (of the seed)*. The *size* of $\mathcal{T}$ is measured by the number of tile types employed, that is, $|T|$. According to the following general rule that all RTASs obey, it tiles the first quadrant delimited by the seed:

RTAS tiling rule: A tile $t \in T$ can attach to an assembly α at position (x, y) if

1. $\alpha(x, y)$ is undefined,
2. both $\alpha(x-1, y)$ and $\alpha(x, y-1)$ are defined,
3. $t(\mathtt{W}) = \alpha(x-1, y)[\mathtt{E}]$ and $t(\mathtt{S}) = \alpha(x, y-1)[\mathtt{N}]$.

The attachment results in a larger assembly β whose domain is $\mathrm{dom}(\alpha) \cup \{(x, y)\}$ such that for any $(x', y') \in \mathrm{dom}(\alpha)$, $\beta(x', y') = \alpha(x, y)$, and $\beta(x, y) = t$. When this attachment takes place in the RTAS $\mathcal{T}$, we write $\alpha \to_1^{\mathcal{T}} \beta$. Informally speaking, the tile t can attach to the assembly α at (x, y) if on α, both $(x-1, y)$ and $(x, y-1)$ are tiled while (x, y) is not yet, and the west and south glues of t match the east glue of the tile at $(x-1, y)$ and the north glue of the tile at $(x, y-1)$, respectively. This implies that, at the outset, (0, 0) is the sole position where a tile may attach. For those who are familiar with the aTAM [45], it should be straightforward that an RTAS is a temperature-2 tile assembly system all of whose glues are of strength 1.

Example 1 See Fig. 1 for an RTAS with 4 tile types that self-assembles the binary counter pattern. To its L-shape seed shown there, a black tile of type (1, 1, 0, 0, black) can attach at (0, 0), while no tile of other types can due to glue mismatches. The attachment makes the two positions (0, 1) and (1, 0) attachable. Tiling in RTASs thus proceeds from south-west to north-east *rectilinearly* until no attachable position is left.

The set $\mathcal{A}[\mathcal{T}]$ of *producible* assemblies by $\mathcal{T}$ is defined recursively as follows: (1) $\sigma_L \in \mathcal{A}[\mathcal{T}]$, and (2) for $\alpha \in \mathcal{A}[\mathcal{T}]$, if $\alpha \to_1^{\mathcal{T}} \beta$, then $\beta \in \mathcal{A}[\mathcal{T}]$. A producible assembly $\alpha \in \mathcal{A}[\mathcal{T}]$ is called *terminal* if there is no assembly β such that $\alpha \to_1^{\mathcal{T}} \beta$. The set of terminal assemblies is denoted by $\mathcal{A}_{\Box}[\mathcal{T}]$. Note that the domain of any producible assembly is a subset of $(\{-1\} \cup [w]) \times (\{-1\} \cup [h])$, starting from the seed σ_L whose domain is $\{(-1, -1)\} \cup ([w] \times \{-1\}) \cup (\{-1\} \times [h])$.

A tile set T is *directed* if for any distinct tile types $t_1, t_2 \in T$, $t_1(\mathtt{W}) \neq t_2(\mathtt{W})$ or $t_1(\mathtt{S}) \neq t_2(\mathtt{S})$ holds. An RTAS $\mathcal{T} = (T, \sigma_L)$ is *directed* if its tile set T is directed (the directedness of RTAS was originally defined in a different but equivalent way). It is clear from the RTAS tiling rule that if $\mathcal{T}$ is directed, then it has exactly one terminal assembly, which we call γ. Let γ' be the subassembly of the terminal assembly such that $\mathrm{dom}(\gamma') \subseteq \mathbb{N}^2$, that is, the tiles on γ' did not originate from the seed σ_L but were tiled by the RTAS. Then we say that $\mathcal{T}$ *uniquely self-assembles the pattern* $P(\gamma')$.

The *pattern self-assembly tile set synthesis* (PATS), proposed by Ma and Lombardi [25], aims at computing the minimum size directed RTAS that uniquely self-assembles a given rectangular pattern. The solution to PATS is required to be directed here, but not originally. However, in [15], it was proved that among all the RTASs that uniquely self-assemble the pattern, the minimum one is directed.

To study the algorithmic complexity of this problem on "real size" particle placement problems, a first restriction that can be placed is on the number of colors allowed for the input patterns, thereby defining the k-PATS problem:

k-COLORED PATS (k-PATS)
GIVEN: a k-colored pattern P
FIND: a smallest directed RTAS that uniquely self-assembles P

The **NP**-hardness of this optimization problem follows from that of its decision variant, which decides, given also an integer ℓ, if such an RTAS is implementable using at most ℓ tile types or not. In the rest of this paper, we use the terminology k-PATS to refer to this decision problem, unless otherwise noted.

3 2-PATS is NP-hard

We will prove that PATS is **NP**-hard for binary patterns (2-colored patterns) by providing a polynomial-time reduction from *monotone satisfiability with few true variables* (M-SAT) to (the decision variant of) 2-PATS. In M-SAT we consider a number k and a Boolean formula F in conjunctive normal form *without negations* and ask whether or not F can be satisfied by only allowing k variables to be true. In fact, M-SAT is just an alias of the well-known SET-COVER problem (for its **NP**-completeness, see, e.g., [13], where it is rather called MINIMUMCOVER). Nonetheless, interpreting the SETCOVER rather as a variant of SAT enables us to naturally adopt know-how accumulated in the existing proofs (e.g., [7, 21, 39]) for the **NP**-hardness of PATS and k-PATS, many of which employed SAT or its variants as a source reduction.

Given an instance of M-SAT, which is a formula F and an integer k, we reduce it to a binary pattern $P_{k,F}$ such that a directed RTAS with $\ell = 13$ or less tile types self-assembles $P_{k,F}$ if and only if the answer to the M-SAT instance is yes, that is, F can be satisfied with exactly k true variables.

3.1 The gadget pattern G

We design the pattern $P_{k,F}$ so as to incorporate, as a subpattern, a gadget pattern G shown in Fig. 2. As formally stated in Lemma 1 below, the gadget pattern G has the property that among all the tilesets of size at most 13, exactly one (up to isomorphism) can be employed in a directed RTAS to assemble G, and thus any pattern with G as a subpattern has the same property. Let T be this tileset, shown in Fig. 3. Lemma 1 is verified by an exhaustive search by a computer program; see Sects. 4 and 5. All the other parts of our proof of Theorem 1 are manually checkable.

Lemma 1 *If a directed RTAS whose tileset consists of 13 or less tile types self-assembles the gadget pattern G in Fig. 2, then its tileset is isomorphic to T.*

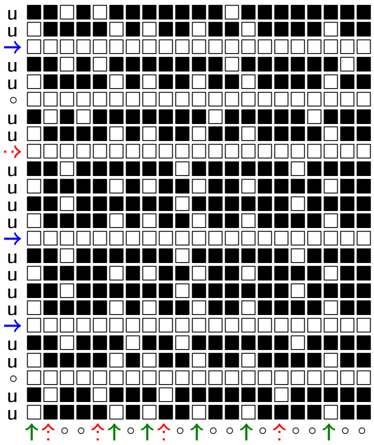


Fig. 2 The binary gadget pattern G can only be assembled by using the tile set T (or one isomorphic to it) unless 14 or more tile types are available. To self-assemble G using T one has to use the glues on the L-shape seed as indicated on the bottom and left. For performance purposes, the bottom row in the pattern was not included in the computerized search; however, because uncovering rows (i.e., rows with horizontal glues u) appear in pairs, we add the bottom row here for clarity.

Due to this property of G, in order to decide the reduced 2-Pats instance $(P_{k,F}, 13)$, it suffices to decide whether a directed RTAS with tileset T self-assembles $P_{k,F}$ or not. This is equivalent to finding an L-shape seed σ_L such that the directed RTAS (T, σ_L) self-assembles $P_{k,F}$. A subtlety of our proof comes from the fact that neither F nor k influence the optimal number of tile types that can assemble $P_{k,F}$ if F is satisfiable.

3.2 The tileset T

The tileset T works as an M-SAT verifier when being used by a directed RTAS. It contains 11 white tile types and 2 black ones; see Fig. 3.

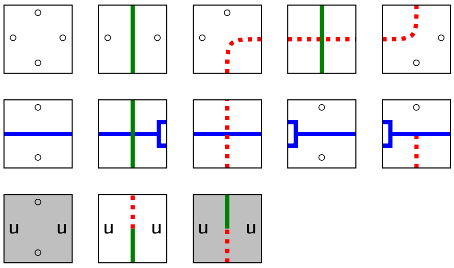


Fig. 3 The tileset T: The background depicts the color of each tile type and the labels and signals depict the glues (i.e., the glue on a side is equivalent to the label or signal on that side, and the colored signals do not actually appear on the tiles). We refer to the tile types with a gray background as the black tile types. For better visibility in printouts, the red signals are dotted; blue and green signals can easily be distinguished as blue signals run only horizontally while green signals run only vertically.

The RTASs given by tileset T verify a given M-SAT instance and present its verification visually on its resulting assembly by "propagating signals" of three kinds (red, green, and blue) via glues from bottom-left to top-right (as the tiles attach in that ordering) and letting them interact with each other. An important fact, that justifies the "signal" vocabulary, is that these signals never fork, that is, in all the tile types of T, if a signal of type s appears on a west or south glue of a tile $t \in T$, it appears on at most one other side, which is either the east or the north side of t.

We interpret the glues in tile set T as follows. Ten of the white tile types (first and second rows in Fig. 3) simulate three types of signals and their interactions. Recall that in the RTAS, growth begins from an L-shape seed and proceeds strictly up and to the right. Therefore, as tiles are added by matching the signals on their bottom and/or left sides, we can think of them as passing the signals to their output (i.e., top and/or right) sides, as indicated by the colored lines showing the signals across each tile. These signals can necessarily, due to the ordering of growth of the assembly and the definitions of the tile types, move only up, right, up and right, or terminate. The signals propagate as follows:

1. blue signals propagate left to right,
2. green signals propagate from bottom to top, and
3. red signals propagate diagonally, bottom left to top right in a wavelike line.

When any two of the signals meet, they simply cross over each other, while the red signal is displaced upwards or rightwards when crossing a blue or green signal, respectively. However when a blue signal crosses a green signal immediately before encountering a red signal, the red signal is destroyed. In order to recognize this configuration, the blue signal is *tagged* when it crosses a green signal; in Fig. 3, the tagging is displayed by the fork in the blue signal. Let us stress that the signals are encoded in the glues of the tiles, and not (at least directly) in their colors.

The other three tile types, called *uncovering tiles*, all with horizontal glues of type u, are used to start rows called *uncovering rows*. A major challenge of the reduction is that we cannot force our signals to appear directly in the pattern, because we have only two colors. Instead, we start these uncovering rows, and make the signals appear in the pattern by their effects on these rows. More specifically, rows with horizontal u glues are always used in pairs. Table 1 shows which pattern of two stacked colors corresponds to which uncovering tiles and signal. Note that by the definition of the tile set, it's impossible for two signals to be received in the same column. Moreover, blue signals are not uncovered, since they never reach these rows. Green (resp. red) signals switch to red (resp. green) in the first uncovering row, but they switch back to their original state in the second uncover row. This allows the enforcement of the encoding of the three possible values of signals (no signal, green signal, or red signal) with exactly two colors. In our construction, uncovering rows always appear in pairs in order to ensure that the original state of each signal is reestablished after passing through a pair of uncovering rows.

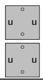
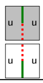
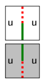
pattern	unique tiles of T	signal from below
■ ■		no signal
■ □		green
□ ■		red

Table 1 Color pairs in the uncovering rows and their corresponding signals

The interactions of the tiles in the tileset T and, in particular, the signals which are encoded in the glues of the tiles are illustrated in Figure 4. It shows an example subassembly which represents the formula $F = (x \vee y) \wedge (y \vee z)$ and $k = 1$, without the gadget part. A more extensive example of a tile assembly with tileset T, shown in Fig. 7 in Sect. 3.3, is the tiling of the gadget pattern G together with some initialization rows.

We have already given an intuition how the red signals progress through the pattern, and it is also clear that the green signals always progress upwards from one glue to the next and blue signals progress rightwards from one glue to the next. The following lemma formalizes the progression of red signals through green and blue signals. It will play an important role in encoding the

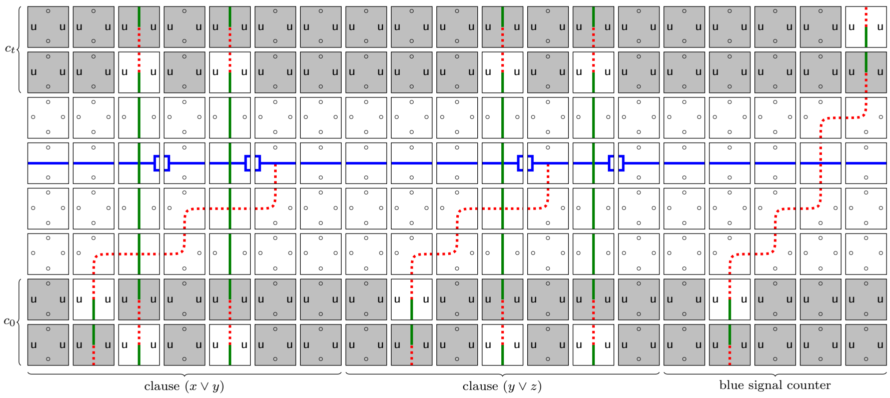


Fig. 4 Subpattern of $P_{k,F}$ for the formula $F = (x \vee y) \wedge (y \vee z)$ with $k = 1$: The position of the blue signal represents the satisfying variable assignment $\phi_y = 1$. Only the subpattern which encodes F is shown, the gadget pattern and the areas needed to initialize the gadget pattern are omitted here. The different subpatterns shown here are explained in the proof of Theorem 1.

integer k of the given M-Sat instance into the pattern $P_{k,F}$, as well as in proving how red signals can be destroyed; see Fig. 5 for an illustration.

Lemma 2 *Let the south glue of a tile in position (x, y) be a red signal. If this red signal progresses up-/rightwards to the south glue of a tile in position (x', y') (hence, the tile at position (x', y') is not considered to be crossed) while crossing i green signals, and j blue signals and/or uncovering rows (where uncovering rows have to appear as pairs, i.e., two consecutive rows), then*

$$x' - x - i = y' - y - j.$$

Proof In every row where the red signal crosses a blue signal, the red signal remains at its horizontal position. When the red signal passes a pair of uncovering rows, the red signal is turned into a green signal and then back into a red signal while remaining at its horizontal position. Thus, only in the $y' - y - j$ rows without blue signal or uncovering glue u the red signal moves rightwards. In each of these rows, we move one positions rightwards plus one position for every green signal that is crossed on the total way. We conclude that $x' = x + i + (y' - y - j)$. □

3.3 Proof of **NP**-hardness

The intuition of the construction of the pattern $P_{k,F}$ and its assembly is that on the vertical arm of the seed, variables $x_0, x_1, \ldots, x_{n-1}$ are encoded successively, by the presence of a blue signal if the corresponding variable is set to 1, and a tile with no signal otherwise. Each clause of F is, on the other hand, encoded on the horizontal arm of the seed as a red signal followed by precisely spaced

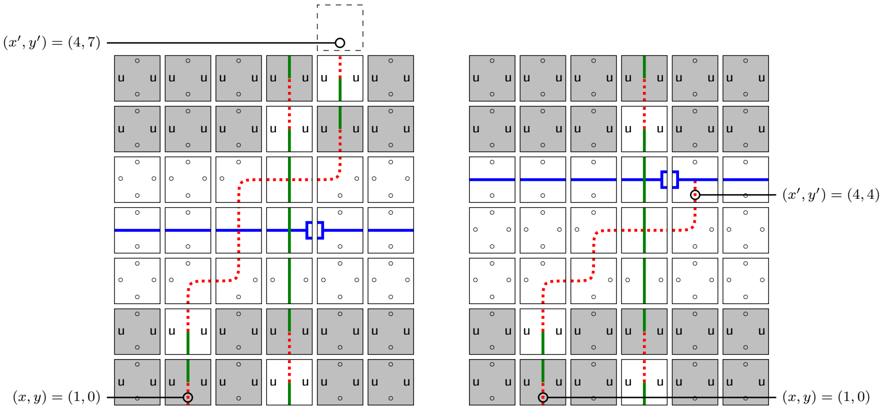


Fig. 5 Example interactions of the signals in the tile set T with uncovering of the configurations: On the left side the red signal can pass through the pattern while the red signal on the right side is destroyed. Note that the position of the blue signal, which is hidden in the horizontal glues, controls whether or not the red signal is destroyed. The marked coordinates (x, y) and (x', y') are the ones defined in Lemma 2.

green signals (intervals between these signals specify which variables are in the clause); see Fig. 4.

For instance, in Fig. 5, the red signal on the left makes it through (i.e., it is not stopped by a tagged blue signal) and appears in the top uncovering rows, while the one on the right does not. The reason for the red signal being stopped on the right, is that the horizontal spacing between the red and the green signal is "compatible" with the vertical location of blue signal. This compatibility of blue, green, and red signals corresponds to a variable in a clause, represented by the red and green signal, which is set true in the variable assignment, represented by the blue signal. More generally, the absence of red signals on the top uncovering rows c_t means that all the clauses have been satisfied, and the presence of a red signal means that at least one clause could not be satisfied by the assignment. Additionally, note that the positions of blue signals, encoding which variables are set to true in a variable assignment of the M-Sat instance, do not appear in the pattern at all, since they travel only through white tiles.

The part of Fig. 4 which is labeled the "blue signal counter" specifies the number k of true variables in a satisfying variable assignment for F. Note that the horizontal movement of the red signal from rows c_0 to rows c_t determines the number of blue signals that appear in the white rows in between c_0 and c_t; see Lemma 2.

Let us now prove Theorem 1, the **NP**-hardness of 2-Pats.

Proof (Proof of Theorem 1) Let $k \in \mathbb{N}$ and F be a set of m clauses which is an instance of M-Sat. For convenience, we assume that F is defined over the n variables $V = \{0, 1, \ldots, n-1\}$. We design a pattern $P_{k,F}$ based on k and F such that $P_{k,F}$ can be self-assembled with no more than 13 tile types if and only if F is satisfiable with only k positive variables. Pattern $P_{k,F}$, schematically presented in Fig. 6, contains a pair c_0 of uncovering rows which we call the *initial configuration*, and a pair c_t of uncovering rows called the *target configuration*. c_0 and c_t are separated by $k+n$ completely white rows. The gadget pattern G is appended in the top left corner of the pattern and is separated by 11 white rows from the target configuration. The area to the right of G does not really matter to the reduction, but it needs to correspond to a valid pattern producible by T. We will describe how to generate it later. Note that we are only interested in directed RTASs whose tilesets are of size 13 or less. Since $P_{k,F}$ includes G as subpattern, Lemma 1 allows us to focus only on directed RTASs whose tileset is T.

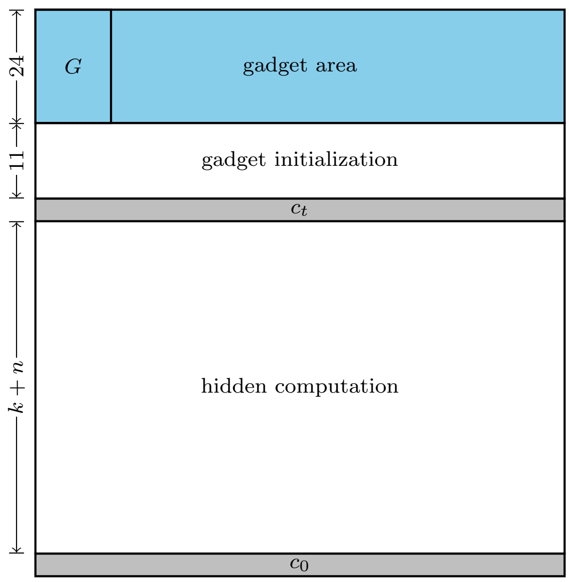


Fig. 6 The pattern $P_{k,F}$, consisting of $k+n+39$ rows. Non-white areas consist of black and white patterns, while white areas consist only of white tiles.

The target and initial configurations are represented by two rows of black or white pixels and do not contain a pair of white pixels above each other. In other words, they are sequences over the three-letter alphabet $\{\substack{\blacksquare\\ \blacksquare}, \substack{\blacksquare\\ \square}, \substack{\square\\ \blacksquare}\}$. Recall that in the assemblies produced by tileset T, $\substack{\blacksquare\\ \blacksquare}$ encodes the absence of a signal, $\substack{\blacksquare\\ \square}$ encodes a green signal, and $\substack{\square\\ \blacksquare}$ encodes a red signal; see Table 1. The target and initial configuration rows are

$$c_t = w_G \begin{smallmatrix}\blacksquare&\blacksquare\\ \blacksquare&\blacksquare\end{smallmatrix} w_0 \begin{smallmatrix}\blacksquare&\blacksquare\\ \blacksquare&\blacksquare\end{smallmatrix} w_1 \cdots \begin{smallmatrix}\blacksquare&\blacksquare\\ \blacksquare&\blacksquare\end{smallmatrix} w_{m-1} v_{t,k}$$
$$c_0 = w_G \begin{smallmatrix}\blacksquare&\square\\ \blacksquare&\blacksquare\end{smallmatrix} w_0 \begin{smallmatrix}\blacksquare&\square\\ \blacksquare&\blacksquare\end{smallmatrix} w_1 \cdots \begin{smallmatrix}\blacksquare&\square\\ \blacksquare&\blacksquare\end{smallmatrix} w_{m-1} v_{0,k}.$$

where w_G, $v_{0,k}$, $v_{t,k}$, $w_0, \ldots, w_{m-1}$ are words over pixel pairs $\{\genfrac{}{}{0pt}{}{\blacksquare}{\blacksquare}, \genfrac{}{}{0pt}{}{\blacksquare}{\square}, \genfrac{}{}{0pt}{}{\square}{\blacksquare}\}$. We let

$$w_G = \begin{smallmatrix} \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \square & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \square & \blacksquare & \square & \blacksquare & \blacksquare & \square & \blacksquare & \blacksquare & \square & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \square & \blacksquare & \blacksquare \end{smallmatrix}$$

$$v_{0,k} = {\genfrac{}{}{0pt}{}{\blacksquare}{\blacksquare}}^{k} \genfrac{}{}{0pt}{}{\square}{\blacksquare} {\genfrac{}{}{0pt}{}{\blacksquare}{\blacksquare}}^{n}$$

$$v_{t,k} = {\genfrac{}{}{0pt}{}{\blacksquare}{\blacksquare}}^{k+n} \genfrac{}{}{0pt}{}{\square}{\blacksquare}$$

Note that w_G encodes the green signals which appear in the gadget pattern G. For $0 \leq i < m$ and $j \in V$, let $x_{i,j} = \genfrac{}{}{0pt}{}{\blacksquare}{\blacksquare}$ if variable j does not appear in clause C_i, and $x_{i,j} = \genfrac{}{}{0pt}{}{\blacksquare}{\square}\genfrac{}{}{0pt}{}{\blacksquare}{\blacksquare}$ if variable j appears in C_i. Recall that $\genfrac{}{}{0pt}{}{\blacksquare}{\blacksquare}$ represents "no signal" and $\genfrac{}{}{0pt}{}{\blacksquare}{\square}\genfrac{}{}{0pt}{}{\blacksquare}{\blacksquare}$ represents "a green signal followed by no signal". Then, let $w_i = x_{i,0} x_{i,1} \ldots, x_{i,n-1}$. Furthermore, $v_{0,t}$ and $v_{t,k}$ encode a *blue signal counter* that contains only one red signal.

In this way, the horizontal arm of the seed and the first two glues of the vertical arm (counting from the bottom) cause c_0 to assemble, which represents the clauses of F. Note that c_0 uniquely determines the glue sequence on the horizontal and the first two glues of the vertical arm. Then, the next $k+n$ glues on the vertical arm of the seed, that encode which of the k variables are set to true, act as input along with c_0 for the growth of the section called *hidden computation* in Fig. 6. The tiles of this section are completely white, but when its assembly completes, then the north glues of the tiles in row $k+n+2$ encode whether or not each clause is satisfied in the hidden computation. The next two rows are a pair of uncovering rows c_t which can only correctly form the pattern $P_{k,F}$ if F was satisfied by the k variables set to true.

Above c_t are 11 *gadget initialization* rows used to initialize the correct growth of the gadget pattern G. They are required so that the red signals encoded in the corresponding 11 glues of the vertical arm of the seed can move the necessary distances to the right and serve as input for G. The top row of the gadget initialization section exposes a combination of these red signals, plus green signals that have passed upward from the horizontal arm of the seed. Thus, this row is composed as follows:

$$W_G \uparrow \circ W_0 \uparrow \circ W_1 \cdots \uparrow \circ W_{m-1} (\uparrow)^{k+n+1}$$

where W_G represents the glues on the bottom border of G, pictured on Fig. 2, and for all $0 \leq i < m$, the glue sequence W_i is defined in a way similar to w_i: for all $j \in V$, let $X_{i,j} = \uparrow \circ$ if j appears in C_i, and $X_{i,j} = \circ$ otherwise. Then, let $W_i = X_{i,0} X_{i,1} \ldots X_{i,n-1}$. The glue sequence W_G, which is the south border of G, can be initialized with the 11 rows below it: see Fig. 7 for the exact way to do this. Note that the red signal which is part of the blue signal counter will run out of the pattern and will not appear in the glue sequence anymore. To the right of the gadget pattern is the *gadget area* which is the pattern that the tile set T self-assembles given as left glues the signals that leave the gadget pattern G on its right, and $\uparrow \circ W_0 \uparrow \circ W_1 \cdots \uparrow \circ W_{m-1} (\uparrow)^{k+n+1}$ as bottom glues.

Note that the width of pattern $P_{k,F}$ is in $O(n \cdot m)$, its height is $k+n+39$, and that it can clearly be computed from k and F in polynomial time as it requires only a simple encoding of the clauses of F, the counter for k, and the constant information of the pattern G. See Fig. 4 for the conversion of a formula with three variables and two clauses into a pattern where the gadget pattern, gadget area and its initialization parts are omitted for the sake of simplicity.

Recall that, since $P_{k,F}$ includes G as a subpattern, and by Lemma 1, $P_{k,F}$ can be assembled with a tileset of at most $\ell = 13$ tile types if and only if $P_{k,F}$ can be assembled by tileset T (or a tileset isomorphic to T). To conclude the proof, we show that $P_{k,F}$ can be assembled by T if and only if F is satisfiable with exactly k true variables.

If (F,k) is satisfiable, then $P_{k,F}$ can be assembled by T: First, consider the upper part of the pattern $P_{k,F}$, above and including the two rows c_t. The rows c_t can be assembled by using the uncovering tile types, given the correct glues on the north border of the hidden computation. The 11 white gadget initialization rows, followed by the 24 rows of G, can be assembled by T from the north glues of w_G in c_t and with the glues on the vertical arm of the seed as shown in Fig. 7; from bottom to top these are the glues:

$$\mathsf{u}^2 \textcolor{red}{\dashrightarrow} \circ^4 \textcolor{red}{\dashrightarrow} \circ \textcolor{red}{\dashrightarrow} \circ^2 \textcolor{red}{\dashrightarrow} \mathsf{u}^2 \circ \mathsf{u}^2 \textcolor{blue}{\rightarrow} \mathsf{u}^4 \textcolor{blue}{\rightarrow} \mathsf{u}^4 \textcolor{red}{\dashrightarrow} \mathsf{u}^2 \circ \mathsf{u}^2 \textcolor{blue}{\rightarrow} \mathsf{u}^2 \,.$$

The first 13 glues in this sequence together with the green signals in w_G create the south input of the gadget pattern G, allowing the gadget pattern to self-assemble in the top left corner of $P_{k,F}$. Furthermore, the gadget area is designed to be exactly the pattern that T self-assembles given the signals that leave the gadget pattern and the green signals in c_t.

As a side note, other sequences might yield the same assembly above c_t. The only important point, used in the other direction of the reduction, is that G is a subpattern of $P_{k,F}$, forcing any solution tileset with no more than 13 tile types to be a tileset which is isomorphic to T.

Next, we prove that the lower part of the pattern, up to and including the two rows of c_t, can be self-assembled by T if F is satisfiable with exactly k positive variables. Let $\phi \in \{0,1\}^n$ be a satisfying variable assignment for F such that $k = |\{i \mid \phi_i = 1\}|$. We define the glue sequence Z, the $k+n$ glues on the vertical arm of the seed to the left of the hidden computation, by $Z = Z_0 Z_1 \ldots Z_{n-1}$ where for all $0 \le i < n$:

$$\begin{aligned} Z_i &= \overset{\textcolor{blue}{\rightarrow}}{\circ} \text{ (on two rows) if } \phi_i = 1 \\ Z_i &= \circ \text{ (on one row) if } \phi_i = 0. \end{aligned}$$

Note that Z contains n glues $\circ$, and k blue signals. Then, the first $n+k+4$ glues, from bottom to top, of the vertical arm of the seed are $\mathsf{uu}Z\mathsf{uu}$. We define the glues for the horizontal arm of the seed by:

$$\textcolor{green}{\uparrow}(\circ)^4\textcolor{green}{\uparrow}\circ\textcolor{green}{\uparrow}(\circ)^2\textcolor{green}{\uparrow}(\circ)^2\textcolor{green}{\uparrow}(\circ)^4\textcolor{green}{\uparrow}(\circ)^2\circ\textcolor{red}{\dot\uparrow}W_0\circ\textcolor{red}{\dot\uparrow}W_1\cdots\circ\textcolor{red}{\dot\uparrow}W_{m-1}(\circ)^k\textcolor{red}{\dot\uparrow}(\circ)^n$$

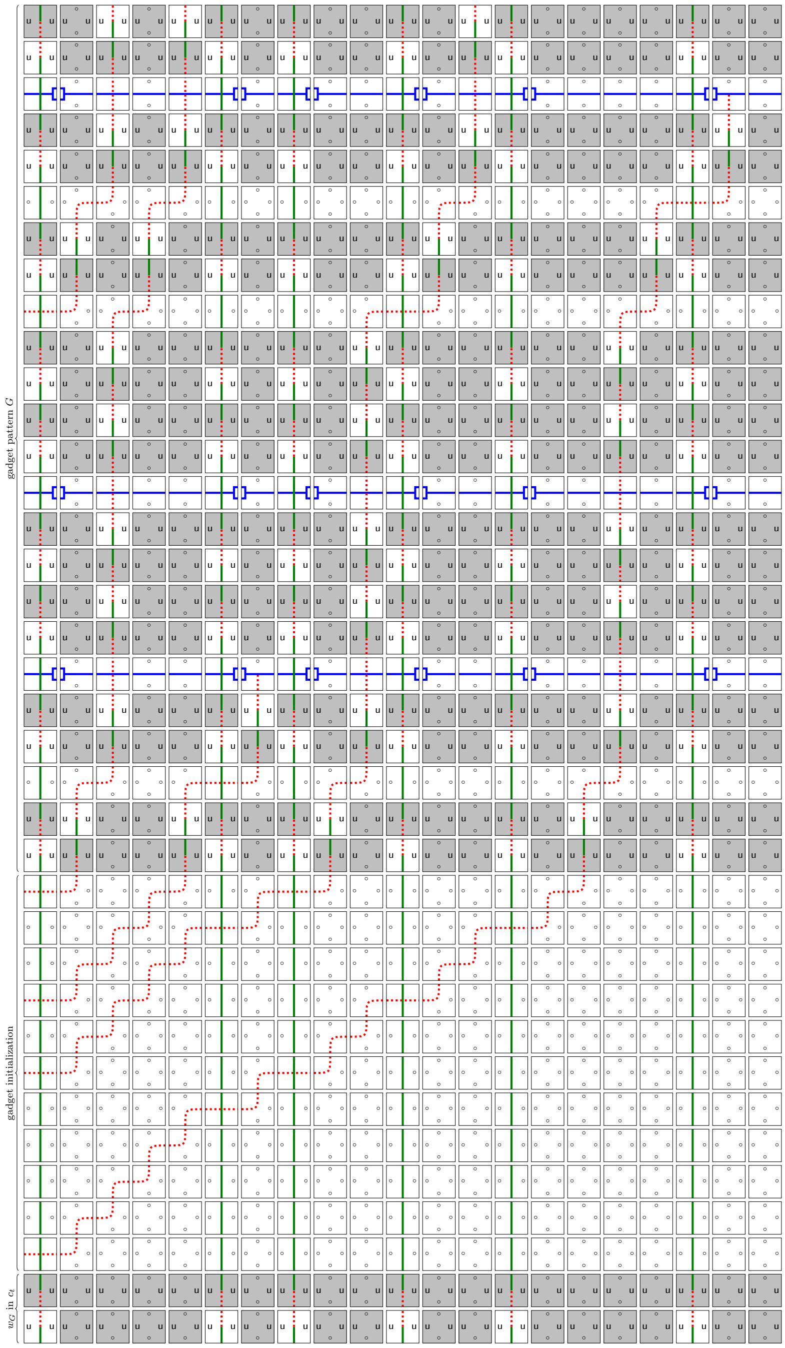


Fig. 7 The gadget pattern with initialization: This subpattern appears in every pattern $P_{F,k}$ in the top left corner and is independent of F and k. It shows how the pattern can be self-assembled above the the first 21 pixel pairs w_G of the two uncovering rows c_t. The eleven white rows are needed in order to initialize the red signals at the south border of G.

The first 21 glues ensure the attachment of w_G, the last $n + k + 1$ glues initialize the blue signal counter which allow the attachment of $v_{0,k}$. The operation of this counter is straightforward from the tileset, as illustrated by the blue signal counter in Fig. 4 and stated in Lemma 2: indeed, the red signal in the counter propagates exactly n columns to the right during the hidden computation rows since it has to pass k blue signals. This allows the right-most part $v_{t,k}$ of c_t to assemble.

The following argument is illustrated in Fig. 8. Let C_i be a clause of F, and $j \in V$ be the smallest variable appearing in C_i such that $\phi_j = 1$. Let a be the number of variables smaller than j that appear in C_i and b the number of variables smaller than j that are true in ϕ. Consider the red signal that starts at the south of the tile in position $(x, 0)$ in the beginning of the encoded clause C_i, immediately to the left of W_i. This signal must cross $a + 1$ green signals and b blue signals plus two uncovering rows before it reaches the south of row $y' = j + b + 3$. After these crossings, by Lemma 2, its horizontal position on the south of row y' is

$$x' = x + (a + 1) + (j + b + 3) - (b + 2) = x + j + a + 2.$$

Note that row y' contains a blue signal since $\phi_j = 1$ and the y'-th glue on the vertical arm is the upper glue of Z_j; and column $x + j + a + 1$ contains a green column because $j \in C_i$ and, therefore, the $(j + a + 1)$-th glue of W_i is the left glue of $X_{i,j}$ which is $\uparrow$. Thus, this red signal stops propagating at this position because it meets a tagged blue signal. As desired, the red signal does not appear in the pair of uncovering rows c_t.

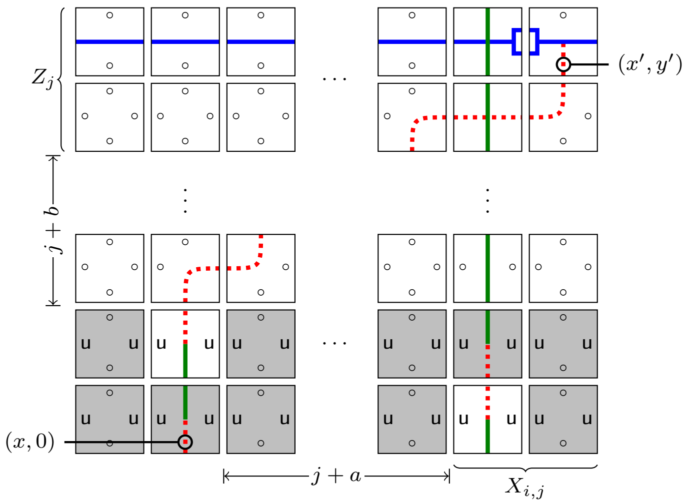


Fig. 8 The red signal which corresponds to a clause C_i is destroyed by the blue and green signals representing the variable $j \in C_i$ with $\phi_j = 1$. The positions $(x, 0)$, (x', y') and the values a, b are explained in the text.

Iterating this argument through all the clauses shows that this part of the pattern can be assembled by T if F is satisfiable with exactly k true variables since no red signals appear in c_t other than the one in the blue signal counter.

If $P_{k,F}$ can be assembled by T, then (F,k) is satisfiable: We will only argue about the pattern between and including c_0 and c_t. The tiles placed on rows c_0 and c_t can only be the three tiles with horizontal u glues since all black tiles on T have horizontal u glues. The only constellations of tiles from T that can assemble pairs of uncovering rows are listed in Table 1. Clearly, the horizontal arm of the seed must have the same glue sequence as described above:

$$\textcolor{green}{\uparrow}(\circ)^4\textcolor{green}{\uparrow}\circ\textcolor{green}{\uparrow}(\circ)^2\textcolor{green}{\uparrow}(\circ)^2\textcolor{green}{\uparrow}(\circ)^4\textcolor{green}{\uparrow}(\circ)^2\circ\textcolor{red}{\uparrow}W_0\circ\textcolor{red}{\uparrow}W_1\cdots\circ\textcolor{red}{\uparrow}W_{m-1}(\circ)^k\textcolor{red}{\uparrow}(\circ)^n.$$

Let Z be the sequence of $n+k$ glues (from bottom to top) left of the hidden computation on the vertical arm of the seed that is used to assembles $P_{k,F}$ by T. Clearly, Z does not contain any uncovering row as there are no black tiles in the hidden computation and, if the white uncovering tile covered an entire row, all south glues of this row had to be green signals. If the first glue of Z is a blue signal, then it cannot block any red signals; otherwise, green signals would be needed immediately before the start points of red signals, which is incompatible with the definition of c_0. Furthermore, Z cannot contain a tagged blue signal since there is no tile in T with a tagged blue signal as west glue and a green signal as south glue. Lastly, it is possible that Z contains red signals, but they have to be destroyed before they reach c_t and they have absolutely no effect on any other signals nor on the pattern.

The red signal that appears in the target configuration of the blue signal counter $v_{t,k}$ has to be the red signal that is released in the initial configuration of the blue signal counter $v_{0,k}$ as this red signal is spaced more than $n+k$ columns without green signals apart from any other red signal. Due to the blue signal counter, by Lemma 2, Z must contain exactly k blue signals.

The positions of the blue signals in Z can be decoded into a variable assignment $\phi \in \{0,1\}^n$ similar to the encoding above: for $0 \leq i < n$, we let $\phi_i = 1$ if and only if there is j such that the j-th glue in Z is a blue signal which is preceded by $i+1$ glues from $\{\circ, \textcolor{red}{\dashrightarrow}\}$ in Z (and an arbitrary number of blue signals). Note that $|\{i \mid \phi_i = 1\}| \leq k$ because there are k blue signals in Z, and two (or more) consecutive blue signals only contribute one true variable and a blue signal in the first position of Z does not contribute a true variable at all.

Let us prove next that every clause in F is satisfied by ϕ. This argument is the reverse of the argument used before and Fig. 8 serves as illustration again. Consider a clause C_i and its encoding as the glue sequence $\circ\textcolor{red}{\uparrow}W_i = \circ\textcolor{red}{\uparrow}X_{i,0}X_{i,1}\cdots X_{i,n-1}$ on the horizontal arm which is enforced by c_0. Let x be the horizontal position of the red signal of this encoding in the first row of $P_{k,F}$. Because no red signal appears in c_t, except the one in the blue signal counter, the red signal in the glue encoding of C_i has to be destroyed at some point in the hidden computation. The green signal and the blue signal that cross each other immediately before the thus tagged blue signal destroys this red signal are said to *verify* the clause C_i. By the numbers of green signals and $\mathtt{u}$ glues in W_i it is clear that the green signal that verifies C_i has to be emitted in W_i. Let $j \in V$ such that the green signal emitted in $X_{i,j}$ verifies C_i, and let a be the number of variables in C_i that are smaller than j; this means the green signal that verifies C_i covers the column $x+j+a+1$ and the red signal gets destroyed in column $x' = x+j+a+2$. Let b be the number of blue signals that the red signal crosses before it meets the blue signal that verifies C_i. By Lemma 2, the red signal is destroyed in row

$$y' = (b+2) + (x+j+a+2) - x - (a+1) = j+b+3.$$

Since the red signal moves past $j+1$ rows without signal, ϕ_j is true in the variable assignment ϕ that we decoded from Z. Furthermore, since $X_{i,j}$ contains a green signal, we have $j \in C_i$. This concludes the proof that ϕ satisfies the clause C_i.

Using this argument for all the clauses in F, we obtain that if $P_{k,F}$ can be self-assembled by T, then F can be satisfied by a variable assignment with at most k true variables. Due to the monotone nature of F, this also implies that F can be satisfied using exactly k true variables. $\square$

4 Programmatic search for the minimal tile set

We present our algorithm for finding all tile sets of a given size ℓ which self-assemble a given k-colored pattern on a rectangle (in our case $\ell = 13$ and $k = 2$). In Sect. 4.1, we show that it is sufficient to generate all *valid tile assemblies* for the given pattern, which use at most ℓ tile types, rather than generating all tile sets with all possible L-shape seeds. Next, we present the algorithm which generates these valid tile assemblies and the corresponding tile sets in Sect. 4.2 and discuss several methods which we implemented to speed up the algorithm. Lastly, we discuss the parallel implementation and the performance of the algorithm in the two programming languages C++ (Sect. 4.3) and Ocaml (Sect. 4.4).

4.1 RTASs defined by assemblies

Consider the k-colored pattern $P\colon [m] \times [n] \to [k]$. Recall that for a tile set $T = \{t_0, \ldots, t_{\ell-1}\}$ a terminal assembly (without the seed structure) of P is a mapping $\alpha\colon [m] \times [n] \to [\ell]$ such that

(a) if $\alpha(x,y) = \alpha(x',y')$, then $P(x,y) = P(x',y')$ for all $x, x' \in [m]$ and $y, y' \in [n]$,
(b) $t_{\alpha(x,y)}(\mathtt{S}) = t_{\alpha(x,y-1)}(\mathtt{N})$ for all $x \in [m]$ and $y \in \{1, \ldots, n-1\}$, and
(c) $t_{\alpha(x,y)}(\mathtt{W}) = t_{\alpha(x-1,y)}(\mathtt{E})$ for all $x \in \{1, \ldots, m-1\}$ and $y \in [n]$.

Condition (a) implies that every tile type can only have one color and conditions (b) and (c) ensure that there are no vertical or horizontal glue mismatches in the assembly. An (partial) assembly is a partial mapping $\alpha\colon [m]\times[n]\to_p [\ell]$ which satisfies the three conditions for all positions which are defined in α. Every RTAS $\mathcal{T}=(T,\sigma_L)$ that self-assembles P yields a terminal assembly $\alpha\colon [m]\times[n]\to[|T|]$ by enumerating the tiles in T where the seed is implicitly constituted by the south glues of the tiles $\alpha(x,0)$ for $x\in[m]$ and the west glues of the tiles $\alpha(0,y)$ for $y\in[n]$.

Conversely, every mapping $\alpha\colon [m]\times[n]\to[\ell]$ which satisfies condition (a) yields a (not necessarily directed) tile set $T_\alpha=\{t_0,\dots,t_{\ell-1}\}$ where condition (b) imposes equivalence classes on the vertical glues and condition (c) imposes equivalence classes on the horizontal glues (e.g., the glue $t_{\alpha(0,0)}(\mathtt{E})$ belongs to the same equivalence class as the glue $t_{\alpha(1,0)}(\mathtt{W})$). For each of these equivalence classes we reserve one unique glue label in T_α; in particular, no vertical glue gets the same label as a horizontal glue. Thus, α is a terminal assembly of P for T_α. Next, we show that if α is a terminal assembly of P for a tile set T, then T is a morphic image of T_α; that is, there exists a bijection of tile types $h\colon T_\alpha\to T$, and a morphisms g from the glues of T_α to the glues of T such that for all $t\in T_A$ and $d\in\{\mathtt{N},\mathtt{E},\mathtt{S},\mathtt{W}\}$ we have $c(t)=c(h(t))$ and $g(t(d))=h(t(d))$. Let $T=\{t_0,\dots,t_{\ell-1}\}$ and $T_\alpha=\{s_0,\dots,s_{\ell-1}\}$ be the chosen tile enumerations with respect to the assembly α, then the bijection h is chosen such that $h(s_i)=t_i$ for all $i\in[\ell]$. Since both, T and T_α, have to satisfy (a), we obtain that $c(s_i)=c(h(s_i))=c(t_i)$ as desired. Furthermore, T_α was defined such that it satisfies the minimal requirements for α to be an assignment according to conditions (b) and (c). Because T must also satisfy these two conditions, it is clear that the morphism g can be defined.

Note that the fact that T is a morphic image of T_α implies that if T is a directed tile set, then T_α is directed as well (though, the converse does not necessarily hold). Henceforth, we call an assembly α *valid* if it is terminal and its corresponding tile set T_α is directed. The algorithm that we present next lists all valid assemblies of P together with their corresponding directed tile sets with at most ℓ tile types. Therefore, up to morphic images of these solution tile sets, it lists all directed tile sets which can self-assemble P. Also note that if a directed tile set S is a morphic image of our tile set T shown in Fig. 3, then T and S are isomorphic. This can easily be verified as every tile set which is obtained by combining any two horizontal glues or any two vertical glues in T is an undirected tile set.

4.2 The algorithm

Instead of fully generating every terminal assembly α of the pattern P and then checking whether or not the corresponding tile set T_α is directed, we generate partial assemblies tile by tile while adapting a generic tile set in each step such that it satisfies conditions (a) through (c) from Sect. 4.1. If a tile set T_α which corresponds to an assembly α is not directed, then we do not have to place any further tiles into this assembly because any larger assembly β which contains α as subassembly has a corresponding undirected tile set T_β and, hence, α cannot be completed to become a valid assembly. This procedure can be illustrated in a tree spanning the search space where every node is a partial assembly with corresponding tile set. Its root is the empty assembly (no tiles are placed) whose corresponding tile set consists of $\ell = 13$ tile types with every glue of every tile type unique and all tiles un-colored. Leaves in this tree are either solutions, valid assemblies of P with a corresponding directed tile set, or breakpoints, nodes whose tile sets are not directed.

The tiles are placed according to a tile placing strategy; that is, each position in α has a successor position where the next tile is placed. The correctness of the algorithm does not depend on the tile placing strategy, however, the performance of the algorithm highly depends on this strategy. Our strategy is to keep the area that is covered by tiles as compact as possible. Performance tests on small patterns confirmed that the average depth of paths in the tree spanning the search space is significantly smaller when using our strategy as compared to the naive row-by-row or column-by-column approaches. The ordering of positions is illustrated in Fig. 9, and is intuitively defined by *"the alternative addition of a row and a column"*, starting as shown in the figure. Formally, this amounts to defining a sequence of coordinates $(x_i, y_i)_{n\in\mathbb{N}}$ inductively by $(x_0, y_0) = (0, 0)$ and

$$(x_{i+1}, y_{i+1}) = \begin{cases} (0, y_i + 1) & \text{if } x_i = y_i, \\ (x_i + 1, 0) & \text{if } x_i = y_i - 1, \\ (x_i, y_i + 1) & \text{if } x_i > y_i, \\ (x_i + 1, y_i) & \text{otherwise.} \end{cases}$$

The cases in the formula can be interpreted as follows, from top to bottom: start a new row, start a new column, add one tile to an existing column, add one tile to an existing row. This simplified formula suggests that the pattern has to be a square, but it can also be interpreted as total ordering on all positions in the rectangular pattern P by simply skipping positions which lie outside of P.

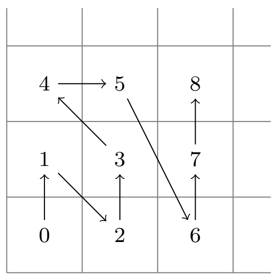


Fig. 9 Order on positions

Let α be a partial assembly in which exactly the first $i \in [n{\cdot}m{-}1]$ positions, according to the tile placing strategy, are covered with tiles and let T_α be the corresponding tile set which we assume to be directed. Therefore, (α, T_α) can be viewed as a node in the tree spanning the search space which is not a leaf. We try out all possible tile types in the empty position $\alpha(x_{i+1}, y_{i+1})$ as follows:

1. If there is a tile type t in the current tile set T which fits (i.e., its glues match those adjacent to the position and its color matches $c(t) = P(x_{i+1}, y_{i+1})$), a tile of that type is placed in $\alpha(x_{i+1}, x_{i+1})$. Note that due to the ordering of tile placements, the adjacent glues (if any) will always be to the west and/or south, ensuring that those are the input sides. If the location is on the bottom (left) edge of the pattern, there will not be an input glue on the south (west).
2. Else:
   (a) For each tile type t which has already been placed somewhere in the assembly and which has color $c(t) = P(x_{i+1}, y_{i+1})$, the west and south glues of t can be changed to match those adjacent to the current position, wherever they occur throughout the current tiles of the assembly (modifying additional tile types as necessary). If the tile set remains directed (i.e., no tile types have the same input glues), then the glue changes are made and t is placed in the current location.
   (b) If the number of tile types which are used in the partial assembly is less than $\ell = 13$, change the glues of one unused type so that it matches those adjacent to the current location, assign the color $c(t) \leftarrow P(x_{i+1}, y_{i+1})$, and place a tile of that type in position $\alpha(x_{i+1}, y_{i+1})$.

Note that this procedure is optimized such that it will not generate two assemblies which are permutations of each other because we do not try several *unused tile types in the same position*. The tree spanning the search space which is defined through this procedure is recursively traversed in a depth-first manner.

If this procedure finds a valid assignment α of P with a corresponding directed tile set T_α, then we output (α, T_α) as solution. As discussed in Sect. 4.1, this algorithm will generate all directed tile sets which can self-assemble P up to morphic images. Both, the Ocaml and C++ version of our program, are parallelized implementations of the algorithm described here.

4.3 Implementation in C++

The C++ code uses MPI[3] as its communication protocol and a simple strategy for sharing the work among the cores. The master process generates a list containing all partial assemblies in which exactly 14 positions are covered by tiles and whose corresponding tile sets are directed. This list contains 271,835 partial assemblies, or *jobs*, in the case of our gadget pattern G from Sect. 3. The master sends out one of these jobs to each of the client processes. Afterwards,

[3] The implementation is Open MPI: http://www.open-mpi.org

the master process only gathers the results of jobs that were finished by clients and assigns new jobs to clients on request. When the list is empty the master sends a kill signal to each client process that requests a new job.

A client process which got assigned the partial assembly α generates all valid assemblies of P which contain α as a subassembly with corresponding directed tile set, using the algorithm described in Sect. 4.2. When one job is finished, possible solutions are transmitted to the master and a new job is requested by the client. In this implementation we did not address the computational bottleneck that emerges when client processes finish the last jobs and then have to idle until the last client process is finished. There is no concept of sharing a job after it has been assigned to a client.

The C++ implementation of our algorithm was run on the cluster `saw.sharcnet.ca` of Sharcnet[4]. The cluster allowed us to utilize the processing power of 256 cores of Intel Xeon 2.83GHz (out of the total 2712 cores) for our computation. In order to minimize the chances of the already unlikely event that undetected network errors influenced the outcome of the computation, our program was run twice on this system, with both runs yielding the same result, namely that the tile set T from Fig. 3 is the only tile set (up to isomorphism) with 13 or less tile types capable of generating the gadget pattern G, thus proving Lemma 1. Each of the computations finished after almost 35 hours using a total CPU time of approximately 342 days. Note that this implies a combined CPU idle time of about 30 days for the clients which we assume to be chiefly caused by the computational bottleneck at the end of the computation. During one computation all the cores together generated over $66 \cdot 10^{12}$ partial tile assemblies.

Later a third run was performed on the Sharcnet cluster `orca.sharcnet.ca` utilizing 256 cores of AMD Opteron 2.2–2.7GHz. In this run the roles of the x- and y-coordinate in the tile placing strategy was swapped, which turned out to reduce the search space by about 7%, yet the computational time was slightly higher due to the different core architecture of the clusters.

4.4 Implementation in Ocaml

The kind of intensive proofs our approach uses has traditionally been proven "rigorously", with consensual proofs, several years after their first publication. This means that some of the latest proofs rely on the simplicity of their implementation to make them checkable. Moreover, proof assistants like Coq are not yet able to provide a fast enough alternative, to verify really large proofs in a reasonable amount of time.

Things are beginning to change, however, and the gap between rigorous and algorithmic proofs is being progressively bridged. The ultimate goal of this research direction is to get rigorous proofs as the first proof of a theorem, even in the case of explorations run on large parallel computers.

[4] https://www.sharcnet.ca/my/systems/show/41

In order to reach this goal for Lemma 1, we wrote a library to be used for additional algorithmic proofs whose size requires a parallel implementation. It also allows for working with different computing platforms, including grids, clusters and desktop computers.

This library, called *Parry*, is available at `http://parry.lif.univ-mrs.fr`.

Remarkably enough for a parallel program, its proof makes no hypotheses on the network, and only relies on the equivalence of Ocaml semantics and its compiled assembly version, as well as on the security of cryptographic primitives. The results of the exploration can be seen at `http://pats.lif.univ-mrs.fr`.

5 The algorithm in OCaml and the proof of Lemma 1

We now discuss the Ocaml implementation of the proof of Lemma 1, and give a proof of this implementation. In order to make this part reusable for other projects, we first wrote a library, called *Parry*, and then wrote a specialized version of it for the 2-Pats problem.

5.1 Global overview of the architecture

Our system is composed of two main components, a "`server`" and a "`client`". The `server` orchestrates the work done by a collection of `clients` by assigning *jobs* (where a job is a current tile set and partial assembly) to each, monitoring their progress, and recording all discovered solutions. The `clients` are assigned jobs by the server and perform the actual testing of all possible tile sets within the fixed size bound (i.e., 13 tile types) to see if they can self-assemble the input pattern. To prove the correctness of the system, we will individually prove the correctness of the `server` and `client`. The main result to be proven for the system is the following:

Lemma 3 *The `server` completes its search if and only if all tilesets of size* ≤ 13 *(up to isomorphism) which can self-assemble the input pattern have been discovered.*

The task of the `server` is to assign and keep track of all jobs which are being explored by the `clients`. Each `client` connects to it to ask for a job assignment. The `server` then replies with an assignment and keeps track of that job in case the `client` crashes, in which case the `server` will be able to detect that (in a way to be discussed) and reassign the job to another `client`. Along with that job, the `server` sends a Boolean indicating whether it expects the job to be re-shared.

The `clients'` messages to the `server` can be of three kinds: "get job" messages, new jobs (in our case, new tilesets and new partial assemblies to be explored), or a "job done" message, to tell the `server` that the job has been completed.

5.2 The implementation

Our strategy to prove the whole system is the following:

- First prove, in Sect. 5.4, an invariant on the server's state, conditioned on hypotheses called *validity* and *fluency* on the clients with a valid RSA signature of their messages (we will also prove that it rejects all other clients).
- Then prove, in Sect. 5.5, that our clients respect the fluency and validity condition, if their worker function (which is the actual implementation of the algorithm described in Sect .4.2), shares the work properly.
- Finally, prove, in Sect. 5.6 that our worker function shares the work properly.

The reason for this organization is to make the proof for the server and client reusable in other applications.

Definition 1 Let T and R be two sets, whose elements are called *jobs* and *results*, respectively. Let $f : T \to 2^R$ be any function. If there is a set $\{t_1, \ldots, t_n\} \subseteq T$ such that $f(t) = \bigcup_{1 \leq i \leq n} f(t_i)$, we call $t_1, \ldots, t_n$ *subjobs* of t.

We say that a task t has been *explored* when $f(t)$ is known.

5.3 How to read the code, and what we prove on it

The language we used to implement this architecture is the functional language Ocaml. Although the syntax of this language may be somewhat surprising at first, the essential points that make our program easier to read, and easy to prove, are:

- Our program makes almost no use of mutable variables: the server state is completely held within a single record variable, and changes are made by nondestructive updates, which allows it to remain in a consistent state after each modification. This is especially important in the server, which makes use of threads.
  All other variables that we use are *non-mutable*, meaning that *new variables* are created whenever a change is needed. For instance, in the client's `place_tile` function, we will see in Sect. 5.6 that new partially assembled patterns are allocated at each tile placement.
  Fortunately, efficient functional data structures exist, that can even make such operations as efficient as in-place updates (this is also due to efficient, specific garbage collectors).
  In particular, `Map` and `Set` will be commonly used in our program.
- In two particular cases, however, we use an Ocaml construct called `ref`. For instance, `let a=ref 0` creates a box called `a`, whose contents is 0. To change the contents to 1, we write `a:=1`. To access the contents of a box, we write `!a`.

- In order for our program to stay "as functional as possible", and therefore as close as possible to a mathematical proof, we tried to use as few global variables as possible: therefore, all core functions depend only on their arguments.
- Surprisingly, we do not need to prove anything about the messages *sent by the server*: any message sent by the protocol can be interpreted as the statement of a lemma of the global proof.
  This makes the following kind of "attacks" possible: an attacker intercepts a "job mission" sent to a client and changes it. The client then starts to work on that job. However, when it sends its "lemma" back to the server, these parts of the proof cannot be used, because they do not correspond to any "proof goal" in the server.

Moreover, many details of our functions need not be proven: we are only interested in the correctness of a program. In particular, we will not prove the efficiency or complexity of our protocol, nor the fact that the server will eventually halt on all runs: the fact that it halts on at least one run is sufficient for Lemma 1 to hold. Moreover, as explained above, proving that a parallel program halts would require additional unproven hypotheses on the network anyway.

5.4 Proving the server

We now proceed to the proof of the server. The main property we use is the fact that the whole state of the server is recorded in a single record datatype called `state`, and defined below.

```
module Server (J : Job) =struct
```

The invariant we are going to prove is on the `state` type: we prove that after any message the server receives, its state contains all the jobs that have not yet been explored, and all the results found during the exploration of already explored jobs.

```
type ongoing = { host :string; key : Cryptokit.RSA.key; job : J.job;
                 start_time : float;
                 last_seen : float }

type state = {
  jobs : JSet.t;
  ongoing : ongoing IntMap.t;
  unemployed :float IntMap.t;
  min_depth :int;
  results : J.result;
  newId :int;
  killings :int;
  solved :int;
  authorized_keys : Cryptokit.RSA.key IntMap.t;
```

```
    reverse_authorized_keys :int StrMap.t;
  }
```

The following function creates an initial state containing the list of jobs and results received as arguments.

```
  let initial_state jobs res =
    (ref {
      jobs = List.fold_left (fun a b → JSet.add b a) JSet.empty jobs;
      ongoing = IntMap.empty;
      unemployed = IntMap.empty;
      min_depth = 0;
      results = res;
      newId = 0;
      killings = 0;
      solved = 0;
      authorized_keys = IntMap.empty;
      reverse_authorized_keys = StrMap.empty
    },Mutex.create ())
```

In order to handle Unix signals properly, we need to avoid these in critical sections, which is done by locking and unlocking mutexes using the two following functions:

```
  let mutex_lock m =
    let _ = Thread.sigmask Unix.SIG_BLOCK
      [Sys.sigint; Sys.sigterm; Sys.sigquit; Sys.sigpipe]
    in
    Mutex.lock m

  let mutex_unlock m =
    Mutex.unlock m;
    let _ = Thread.sigmask Unix.SIG_UNBLOCK
      [Sys.sigint; Sys.sigterm; Sys.sigquit; Sys.sigpipe] in
    ()
```

Definition 2 In a server state `st`, the *current job* of a client is the job registered in the `ongoing` field of `st`.

Definition 3 We call a client *valid* if, at the same time:

1. Its `NewJobs` messages contain all the results in subjobs of its current job that have been completely explored, and the subjobs of its current job that have not been completely explored, divided into three fields: the results it has found, its next current job, and other subjobs.
2. It does not send a `JobDone` message before the task representing its current job is completely explored.

The main function, `answer`, keeps track of the clients. We now prove the following Lemma:

Lemma 4 *If* `st` *is a state of the server containing (in the union of* `job st` *and* `ongoing st`*) jobs representing all the tasks that have not yet been explored, and for any job* `j`*,* `j` *and* `kill j` *represent the same task, then for any message* `m` *sent by a valid client, all values of* `host` *and* `time`*,* `answer host time st m` *(the Ocaml syntax for "the value of function* `answer`*, called with arguments* `t`*,* `host`*,* `st` *and* `m`*") is a couple* (`st'`,`m'`)*, where* `st'` *is a state of the server containing the roots of all subtrees that have not yet been explored (*`m'` *is the message to be sent to the client).*

Moreover, all results sent by the clients are added to the server state using the `add_result` *function.*

Proof We prove it for all the cases.

```
let answer t host key st msg =match msg with
    GetJob num → (
      try
```

If the client is registered as an "ongoing" job, we can simply send it the job it is supposed to be working on. In this case, the invariant is still maintained, as we do not change its recorded current job (here, we only update the time at which we last saw this client).

```
          let ongoing_job = IntMap.find num st.ongoing in
          if host = ongoing_job.host ∧ key = ongoing_job.key then
            ({ st with ongoing =
                        IntMap.add num
                                    { ongoing_job with last_seen =
t }
                                    st.ongoing },
             Job (false,ongoing_job.job))
          else
            (st,Die)
        with
            Not_found → (
```

Else, client `num` is not in the map of ongoing jobs. Therefore, the call to `IntMap.find` above raises exception `Not_found`, that we are catching here. In this case, if there are no more jobs to be done:

- if there are no more jobs being worked on, we do not modify the state, and we tell the client to stop (with a `Finished` message).
- else, we simply record that client as "unemployed". The next time a client reports its state, it will be asked to share its current job. This does not change the jobs registered in the server's state anyway.

```
          if JSet.is_empty st.jobs then
            if IntMap.is_empty st.ongoing then
              (st, Finished)
            else
              let min_depth =
                IntMap.fold
                  (fun _ ongoing d → min d (J.depth ongoing.job))
                  st.ongoing max_int
              in
              ({st with
                unemployed = IntMap.add num t st.unemployed;
                min_depth =if min_depth = max_int then -1 else min_depth},
               Die)
          else
```

Else, there are still jobs to be done; we pick the one with the smallest depth, using `JSet.min_elt`. Therefore, since `num` is not a member of `ongoing st` (this is the case where no exception is raised), the returned state contains, in the union of its `ongoing` and `jobs` fields, exactly the same jobs as in `st`.

```
            let h = JSet.min_elt st.jobs in
            ({ st with
              jobs = JSet.remove h st.jobs;
              ongoing = IntMap.add num
                                   { host = host;
                                     key = key; job = h;
                                     start_time = t; last_seen =
t }
                                   st.ongoing;
              unemployed = IntMap.remove num st.unemployed },
             Job (false,h))
          )
    )
```

Another message the server can receive is the `NewJobs` message, when clients reshare their work: In this case, the client sends its number `num`, its current job `current`, the new job `next` that it will work on, a list `jobs` of jobs that need to be shared, and a list of results. We can think of this message as equivalent to *"I, valid client* `num`*, hereby RSA-certify that job* `current` *you gave me has subjobs* `next`*,* `jobs`*, and results* `results j`*, and no other subjobs or results."*.

If the client is not registered as an "ongoing job", this message is ignored, the state is not modified, and the client is sent the `Die` message.

```
  | NewJobs (num, current, next, jobs, results) → (
    try
      let ongoing = IntMap.find num st.ongoing in
      if ongoing.host = host ∧ ongoing.job = current then
```

The first case is when the host name and current job match what the server had recorded for that client. Recall our assumption that messages processed by `answer` can only be sent by valid clients. Therefore, this message contains all subjobs of its current job that have not been explored, along with the job it will start working on, and the list of all results that have been found during the exploration of the other subjobs of its current job. Since all these subjobs are stored in the `jobs` field of the state, and the `ongoing` field is updated with the client's new current job, our claim still holds.

```
          ({ st with
            jobs = List.fold_left (fun s x → JSet.add x s) st.jobs jobs;
            ongoing =
              IntMap.add num
                          { ongoing with job = next; last_seen = t }
                          st.ongoing;
            results =
              List.fold_left (J.add_result host) st.results results },
           Ack)
        else
          (st, Die)
      with
          Not_found → (st, Die)
    )
    | JobDone (num, current, results) → (
```

In the case of the `JobDone` message, if the client is not registered as an ongoing job, we do not modify the state. Else, we can safely delete the corresponding job from the state, and add its results to the state's results field: indeed, since we assumed that this message is sent by a valid client, that job has been explored completely. The intuitive version of this message is *"I, valid client* `num`*, hereby RSA-certify that I have explored job* `current` *completely, and that it contains exactly results* `results`*"*.

```
      try
        let ongoing = IntMap.find num st.ongoing in
        if ongoing.job = current then
          ({ st with
            ongoing = IntMap.remove num st.ongoing;
            results =
              List.fold_left (J.add_result host) st.results results;
            solved = st.solved + 1 }, Ack)
        else
          (st, Die)
      with
          Not_found → (st, Die)
    )
    | Alive num → (
```

The last case of `answer` is when the client sends an “Alive” message. In that case, we do not modify the contents of `st.ongoing` nor `st.jobs`: indeed, the only operation we do updates a “time” field of `st.ongoing`, so that the client will not be considered dead. Therefore, our claim still holds.

```
try
  let ongoing = IntMap.find num st.ongoing in
  if ongoing.host = host ∧
    (IntMap.is_empty st.unemployed
     ∨ ¬ (J.sharable ongoing.job)
     ∨ J.depth ongoing.job  >  st.min_depth)
  then
    ({ st with
      ongoing =
        IntMap.add num { ongoing with last_seen = t }
                   st.ongoing },
     Ack)
  else
    (st, Die)
with
    Not_found → (st, Die)
)
```

□

Our next task is to prove `reply`, the network interface to the `answer` function. We first need hypotheses on how this interface works, and especially how the messages are written and read at the ends of the connection.

Definition 4 A client is *fluent* if the messages it sends on the network are of exactly two kinds:

- Messages starting with a single byte with value 255 (or `0xff`), and then components `n` and `e` of the client’s public key.
- Messages starting with a single byte $b < 255$, followed by a message m of type `client_message`, as output by the ocaml builtin function `output_value`, and then the RSA signature, using the key of index b in the server, of the SHA-1 hash of m.
  Additionally, we use b as the index of the client’s public key, as registered by the server.

Lemma 5 *If all the clients that have their public key in* `st.authorized_keys`*, where* `st` *is the state of the server, are valid and fluent, and* `st` *contains all the jobs that have not been completely explored (in the* `ongoing` *and* `jobs` *fields), then so does it after one run of* `reply`*, assuming that* `input_value` $\circ$ `output_value` *is the identity, where* `input_value` *and* `output_value` *are ocaml’s builtin functions.*

Proof We prove this invariant on the code of the `reply` function, which handles every connection to our server.

```
let bufsig0 = String.create (J.signature_size lsr 2)
let bufsig1 = String.create (J.signature_size lsr 3)
let buf = String.create (max 2 Marshal.header_size)

let reply rstate mstate descr host =
  let inch = Unix.in_channel_of_descr descr in
  let ouch = Unix.out_channel_of_descr descr in
```

If the client is fluent, then by Definition 4, there are two cases, depending on the first byte received.

```
  really_input inch buf 0 1;

  if int_of_char buf.[0] = ff₁₆ then (
```

In case the first byte is `0xff`, then the next part of the message should be the the public part of its key, in 2`J.signature_size` bits, by Definition 4.

We receive these bits in variable `bufsig0`, and then check if this public key is registered in the server's state, in the `reverse_authorized_keys` field. If so, we send the client an index number for itself, and the index of its public key, for further communication:

```
    really_input inch bufsig0 0 (J.signature_size lsr 2);

    mutex_lock mstate;
    let i = StrMap.find bufsig0 (!rstate).reverse_authorized_keys in
    let newId = (!rstate).newId in
    rstate := { !rstate with newId = (!rstate).newId + 1 };
    mutex_unlock mstate;

    let open Cryptokit.RSA in
    Marshal.to_channel ouch (newId, i) [];
    flush ouch
  ) else (
```

In this case, the client is sending an actual message. By Definition 4, that message has two parts: a real message, sent using `output_value`, and the RSA signature of an SHA-1 hash of that message. However, receiving this message is not completely straightforward, since we need to receive the full message first in order to check its key. The first step is to fetch the client's public key, as indicated by the message's first byte.

```
    let key =
      let current_state =
        mutex_lock mstate;
        let st = !rstate in
        mutex_unlock mstate;
        st
      in
      IntMap.find (int_of_char buf.[0]) (current_state).authorized_keys
    in
```

Then, we receive the message's header, as output by `output_value`. This header indicates the total length, which allows us to receive the full message in variable `bu`. Finally, we receive the signature, on `J.signature_size` bits, in variable `bufsig1`.

```
really_input inch buf 0 Marshal.header_size;
let size = Marshal.data_size buf 0 in
let buffer = String.create (Marshal.header_size + size) in
String.blit buf 0 buffer 0 Marshal.header_size;
really_input inch buffer Marshal.header_size size;
really_input inch bufsig1 0 (J.signature_size lsr 3);
```

We can now verify the signature sent by the client, and check whether it matches the hash of its message. The implementation we use (in module `Cryptokit`) stores decrypted signatures as a suffix of a constant size string, which is why we compare only that suffix (of variable `unwrapped`) with `hash_bu`.

```
let hash_bu = Cryptokit.hash_string sha buffer in
let rec compare_sig a ia b ib =
  if ia < 0 ∨ ib < 0 then true else
    if a.[ia] = b.[ib] then compare_sig a (ia−1) b (ib−1) else false
in
let unwrapped = Cryptokit.RSA.unwrap_signature key bufsig1 in
if compare_sig
  hash_bu (String.length hash_bu − 1)
  unwrapped (String.length unwrapped − 1)
then (
```

Finally, by assumption, if the signature is correct, then the client is valid and fluent, and therefore, its message was output using `output_value`. We can thus safely fetch it using `Marshal.from_string`, from OCaml's standard library. Then, by Lemma 4, the following call to `answer` maintains the claimed invariant:

```
    let t = Unix.time () in
    mutex_lock mstate;
    let (state′, msg) =
      answer t host key !rstate (Marshal.from_string buffer 0)
    in
    rstate := state′;
    mutex_unlock mstate;

    Marshal.to_channel ouch msg [];
    flush ouch;
  )
)
```

□

A major concern, when writing massively parallel programs on machines connected by a network, is the detection of dead machines. Machines can die because of various physical problems, such as power outages; most of the time, however, processes running on clusters die because they have reached their limit time. To handle this problem, our implementation requires jobs to keep the server informed periodically that they are still alive, by sending message `Alive`.

When they stop sending this message for too long, they are considered dead, and their current job is rescheduled to another machine. This is done by a function called `cleanup`, that we prove now:

Lemma 6 *If* `state` *contains, in the* `ongoing` *and* `jobs` *fields, all the jobs that have not been explored, then so does* `cleanup state`.

Proof In the following function: the state is only modified by partitionning the `st.ongoing` map into two maps `a` and `b`, and adding all the jobs of `a` to the `jobs st` set. Therefore, the set of jobs in the union of `jobs st` and `ongoing st` is not modified.

```
let cleanup state =
  let t = Unix.time () in
  let (a, b) = IntMap.partition
                 (fun k ongoing → (t −.ongoing.last_seen) > J.timeout)
                 state.ongoing
  in
  let unemployed =
    IntMap.filter (fun k t1 → (t −.t1) > J.timeout) state.unemployed
  in
  { state with
    jobs =
      IntMap.fold (fun _ ongoing b → JSet.add ongoing.job b)
                    a state.jobs;
    unemployed = unemployed;
    ongoing = b }
```
□

Finally, connections to our server are processed by a function called `server`, using standard unix functions, or emulations thereof, on other platforms.

Lemma 7 *If:*

- *all tasks that have not been completely explored have job representants in the* `ongoing` *and* `jobs` *fields of the* `state` *argument to* `server`*,*
- *all clients that sign their messages with a private RSA key whose corresponding public key is in the* `state` *variable are valid and fluent,*
- `input_value` $\circ$ `output_value` *is the identity,*

then after any number of messages received by the server, variable `state` *also contains jobs representing tasks that have not been completely explored, in the union of its* `ongoing` *and* `jobs` *fields.*

Proof Everything the server does is calling the functions proved in Lemmas 5 and 6.

5.5 Proving the client

```
open Parry_common
open Cryptokit.RSA

module Client (J : Job) =struct

  type config = { server : Unix.inet_addr; port :int; key : Cryptokit.RSA.key }
  exception EFinished
  exception ReportToServer

  let buf = String.create 1
```

Lemma 8 *The* `sign_and_send` *function sends only one kind of messages on the network, consisting of exactly one byte, strictly smaller than 255, followed by a message m generated with* `Marshal.to_string`*, and the RSA signature of the SHA-1 hash of m.*

Then, `sign_and_send` *returns the server's reply, defaulting to message* `Die` *if an error occurs.*

Proof The following code follows closely this specification. It first computes `bu=Marshal.to_string msg`, then `hash_bu`, the SHA-1 hash of `bu`, and finally the signature `signed` of `hash_bu`, before outputting one byte, `msg` and its signature to the network.

```
  let rec sign_and_send retry key_num key sockaddr msg =
    let sock  =  Unix.socket Unix.PF_INET Unix.SOCK_STREAM 0 in
    let ok, serv_msg =
      try
        let bu = Marshal.to_string msg [] in
        let hash_bu = Cryptokit.hash_string sha bu in
        let signed = Cryptokit.RSA.sign key hash_bu in
        buf.[0] ← char_of_int (min 254 key_num);
        Unix.connect sock sockaddr;
```

The next few lines use OCaml buffered “channels” to send and receive values.

```
        let inch = Unix.in_channel_of_descr sock in
        let ouch = Unix.out_channel_of_descr sock in

        output ouch buf 0 1;
        output ouch bu 0 (String.length bu);
        output ouch signed 0 (String.length signed);
        flush ouch;
```

```
      let x = Marshal.from_channel inch in
      Unix.close sock;
      true,x
    with
        Unix.Unix_error (e,f,g) →
          (Unix.close sock;
           false,Die)
      | _ → false,Die
  in
```

If anything went wrong, and `retry` is `true`, then `sign_and_send` waits one second and resend the message exactly once. Else, it returns the server's reply, defaulting to `Die`

```
  if ¬ ok ∧ retry then
    (Unix.sleep 1; sign_and_send false key_num key sockaddr msg)
  else
    serv_msg
```

□

Lemma 9 *The* `get_nums` *function sends only one kind of messages on the network, consisting of exactly one byte equal to 255 (or* `0xff`*), followed by the public components* `n` *and* `e` *of the client's RSA key.*

It returns a couple (`a`, `b`) *of integers, where* `a` *is the client's index, and* `b` *is the key index assigned by the server, or* `-1` *for both values in cases of errors.*

Proof The following indeed follows closely this specification:

```
let get_nums key sockaddr =
  let sock = Unix.socket Unix.PF_INET Unix.SOCK_STREAM 0 in
  try
    Unix.connect sock sockaddr;
    let inch = Unix.in_channel_of_descr sock in
    let ouch = Unix.out_channel_of_descr sock in
    buf.[0] ← char_of_int ff16;
    output ouch buf 0 1;
    output ouch key.n 0 (String.length key.n);
    output ouch key.e 0 (String.length key.e);
    flush ouch;
    let nums = Marshal.from_channel inch in
    Unix.close sock;
    nums
  with
      e → (Unix.close sock; (−1, −1))
```

□

To prove the remaining functions, we need to introduce the following invariant on their arguments:

Invariant 2 *When the* `cur` *variable is not* `Nothing`*, the* `jobs` *and* `results` *variables contain, respectively, the list of all jobs of the contents of* `cur` *that* *have not been completely explored, and the list of results found during the exploration of all other subjobs of the job in* `cur`*.*

Lemma 10 *Let* `work` *be a function such that, at the same time:*

1. *For all values of* `b`*,* `save` *and* `j`*,* `work b save j` *only calls* `save` *with arguments* `l` *and* `r` *such that* `r` *is the list of all results that have been found, when the subjobs of* `j` *that have not been completely explored are all in list* `l`*.*
2. *For all values of* `b`*,* `save` *and* `j`*,* `work b save j` *returns the list of all subjobs of* `j` *that have not been explored, and the list of all results that have been found in* `j`*, in the remaining subjobs of* `j`*.*
3. *For all values of* `b`*,* `save` *and* `j`*,* `work b save j` *does not communicate with the server.*

Then for all values of `conf`*,* `client conf work` *is a valid and fluent client.*

Proof We first prove fluency: in the `client` function below, the only messages that can be sent on the network are sent by `get_nums` and `sign_and_send`. Therefore, by Lemmas 9 and 8, for all values of `conf`, `client conf work` is fluent.

We now prove that `client conf work` is valid. First, the two conditions of validity are not applicable until the client receives work from the server: indeed, neither `NewJobs` nor `JobDone` messages are sent until then.

```
let rec client conf work =
  let sockaddr = Unix.ADDR_INET (conf.server, conf.port) in
  let num, key_num = get_nums conf.key sockaddr in
  if num < 0 then (
    Unix.sleep 1;
    client conf work
  ) else (
    let continue =ref true in
```

The following is the main client loop. The general scheme of that loop is: until a signal has been received (probably because a user or a scheduler is trying to stop this client), ask the server for some work, using the `GetJob` message. There are three possible answers: `Finished`, `Job` and something else.

```
    while !continue do
      let x = sign_and_send true key_num conf.key sockaddr (GetJob num) in
      match x with
          Finished -> exit 0
```

The first possible answer is `Finished`, which tells the client that the whole exploration is completely finished. In this case, the client simply stops.

```
        | Job (share, j) -> begin
```

The second possible answer is a job from the server. In this case, we claim that each time the `ReportToServer` exception is raised, `saved_results` contains exactly the results found at that step, and saved_jobs contains exactly the remaining jobs at that same step.

Indeed, since we assumed that save can only be called by work with arguments that respect this condition, and only save modifies these two references, then at the two points in the following code where ReportToServer is raised, this condition clearly holds.

Therefore, since NewJobs or JobDone messages can only be sent by catching the ReportToServer exception, conditions 1 and 2 of validity clearly hold.

```
let saved_results =ref [] in
let saved_jobs =ref [j] in
let time =ref (Unix.time ()) in
let save results jobs =
  saved_results := results;
  saved_jobs := jobs;

  if ¬ !continue then raise ReportToServer else
    let t = Unix.time () in
    if t −. !time > J.ping then
      begin
        let m = sign_and_send true key_num conf.key
          sockaddr
          (Alive num)
        in
        time := t;
        match m with
            Ack → ()
          | _ → raise ReportToServer
      end
in
try
  let (a, b) = work save j in
  saved_results := a;
  saved_jobs := b;
  raise ReportToServer
with
    ReportToServer →
      let _ =
        match !saved_jobs with
            [] →
              sign_and_send false key_num conf.key
                sockaddr
                (JobDone (num, j, !saved_results))
          | h :: s →
            sign_and_send false key_num conf.key
              sockaddr
```

```
                    (NewJobs (num, j, h, s, !saved_results))
                in
                ()
            end
            | _ →
```

Finally, the server could reply something else, meaning that there are still other clients working, but it has no job available for the moment. In this case, our client waits some time, and asks for a job again.

```
            Unix.sleep 10
        done
    )
```

Altogether, this implies that our client is fluent and valid.

5.6 The 2-Pats instance

We now proceed to the proof of the instance of Parry that we used for our problem. This module is mostly written in purely functional style, without side effects or mutable variables. Therefore, for performance reasons, the encoding of tiles and positions is not the most naive one: first, tiles are encoded as single integers with five bit fields of `wgl` bits each (where variable `wgl` is 5), where the south and west fields are the two leftmost fields. Functions `withC`, `withN`, `withS`, `withW`, `withE` return a new integer with the color, north, south, west and east fields changed, respectively. Functions `color`, `north`, `south`, `west`, `east` return the color, south, west and east field.

Positions are encoded as two 16 bits fields, where the left field is the x component, and the right one is the y component.

This allows to encode assemblies and tilesets in map data structures, which are purely functional (i.e. non-mutable).

```
module Pats = Parry_client.Client(Job)
open Parry_common
open Pats
open Job
```

Lemma 11 *Given a tileset* `tiles`*,* `isDirected tiles = true` *if and only if* `tiles` *is a directed tile assembly system, that is, no two tiles in* `tiles` *have the same input (i.e. south and west) glues.*

Proof `isDirected` works by traversing the tileset using `IntMap.fold`, with a set accumulator. Let $t_1, \dots t_n$ the successive tiles it sees, and $s_1, \dots, s_n$ the corresponding accumulator values, and s_{n+1} the final accumulator value.

We first prove by induction on i that $s_{i+1} = \{(\mathrm{South}(t_j), \mathrm{West}(t_j)) | j \leq i\}$ if and only if no two tiles, among $t_1, t_2, \dots, t_i$, have the same south and west glues.

Since $s_1 = \emptyset$, this holds for $i = 1$. For $i \geq 2$, by the induction hypothesis, if $(\text{South}(t_i), \text{West}(t_i)) \notin s_i$, then t_i has different south and west glues from all t_j for $j < i$, in which case $s_{i+1} = s_i \cup \{(\text{South}(t_i), \text{West}(t_i))\}$.

Else, two tiles have the same south and west glues; thus, exception `Not_directed` is raised, and `isDirected ts` is `false`.

Therefore, the induction hypothesis holds also for $i + 1$.

By induction, we conclude that if no exception has been raised, s_n is defined, and thus the south and west glues of all tiles in `ts` are different, and `isDirected ts` is `true`.

```
exception Not_directed
let isDirected ts =
  try
    let _ = IntMap.fold (fun _ t s →
      let k = t lsr (3 × wgl) in
      if IntSet.mem k s then raise Not_directed else IntSet.add k s
    ) ts IntSet.empty
    in
    true
  with
      Not_directed → false
```

□

We now proceed to the proof of the `merge` function. Its formal specification of this function is given by the following Lemma:

Lemma 12 *Given a tileset* `ts`*, a color* `c`*, an index into the tileset* `i`*, a pair of north/south glues* `a0_` *and* `b0_`*, and a pair of east/west glues* `a1_` *and* `b1_`*, the function* `merge` *returns a tileset in which the* `i`*th tile of* `ts` *is set to color* `c`*, and all north/south glues in* `ts` *equal to* `max(a0_, b0_)` *are set to* `min(a0_, b0_)` *if this value is strictly smaller than* `mgl` *(and left unchanged else), and all east/west glues in* `ts` *equal to* `max(a1_, b1_)` *are set to* `min(a1_, b1_)` *if this value is strictly smaller than* `mgl` *(and left unchanged else).*

Proof In `merge`, the pair `(a0, b0)` is formed such that `a0 = min(a0_, b0_)` and `b0 = max(a0_, b0_)`. Similarly, the pair `(a1, b1)` is formed such that `a1 = min(a1_, b1_)` and `b1 = max(a1_, b1_)`.

Then, a new tileset is created, by folding through all the tiles of `ts`, adding the modified tiles to an accumulator tileset `ts'`. The modification clearly follows our claim.

```
let merge a0_ b0_ a1_ b1_ i0 c ts =
  let (a0, b0) = if a0_ < b0_ then (a0_, b0_) else (b0_, a0_) in
  let (a1, b1) = if a1_ < b1_ then (a1_, b1_) else (b1_, a1_) in
  IntMap.fold (fun i t ts' →
    let u = if i = i0 then withC t c else t in
    let v = if south u = b0 ∧ a0 < mgl then withS u a0 else u in
    let w = if west v = b1 ∧ a1 < mgl then withW v a1 else v in
    let x = if north w = b0 ∧ a0 < mgl then withN w a0 else w in
```

```
    let y = if east x = b1 ∧ a1 < mgl then withE x a1 else x in
    if y = t then ts′ else IntMap.add i y ts′
  ) ts ts
```

□

The next step of our proof is to prove the core computation, called `placeTile`. We first need to introduce the pattern, hardcoded in the program,

```
let pattern =
  [|[|1; 0; 1; 1; 0; 1; 1; 1; 0; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1; 1|];
    [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|];
    [|0; 1; 1; 1; 1; 0; 1; 0; 1; 1; 0; 1; 1; 0; 1; 1; 1; 1; 0; 1; 1|];
    [|1; 1; 0; 1; 1; 1; 0; 1; 1; 0; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1|];
    [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|];
    [|0; 1; 1; 1; 1; 0; 1; 0; 1; 1; 0; 1; 1; 0; 1; 1; 1; 1; 0; 1; 1|];
    [|1; 1; 0; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1|];
    [|0; 1; 1; 1; 1; 0; 1; 0; 1; 1; 0; 1; 1; 0; 1; 1; 1; 1; 0; 1; 1|];
    [|1; 1; 0; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1|];
    [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|];
    [|0; 1; 1; 1; 1; 0; 1; 0; 1; 1; 0; 1; 1; 0; 1; 1; 1; 1; 0; 1; 1|];
    [|1; 1; 0; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1|];
    [|0; 1; 1; 1; 1; 0; 1; 0; 1; 1; 0; 1; 1; 0; 1; 1; 1; 1; 0; 1; 1|];
    [|1; 1; 0; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1|];
    [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|];
    [|0; 1; 1; 1; 1; 0; 1; 0; 1; 1; 0; 1; 1; 0; 1; 1; 1; 1; 0; 1; 1|];
    [|1; 0; 1; 0; 1; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1; 1; 0; 1; 1; 1|];
    [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|];
    [|0; 1; 1; 1; 1; 0; 1; 0; 1; 1; 0; 1; 1; 0; 1; 1; 1; 1; 0; 1; 1|];
    [|1; 1; 0; 1; 0; 1; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1; 1; 1; 0; 1|];
    [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|];
    [|0; 1; 1; 1; 1; 0; 1; 0; 1; 1; 0; 1; 1; 0; 1; 1; 1; 1; 0; 1; 1|];
    [|1; 1; 0; 1; 0; 1; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1; 1; 1; 1; 1|]|]
```

Finally, the core of our algorithm is the following recursive function, `placeTile`, which moves through all locations in the pattern in the ordering shown by Figure 9 and places tiles from the current tile set (often modifying the tile set, too) as long as it is able to. By making recursive calls which attempt all possibilities, it ensures that the full set of possible tile sets (up to isomorphism) is explored and returns exactly those which self-assemble the given pattern. The arguments to `placeTile` are:

1. `save`: a function that we have explained in Lemma 10, that `placeTile` can use to "save" intermediate results in case it is asked to reshare, or killed (for instance if it runs on a cluster).
2. `results`: a list of results that have been found so far.
3. `jobs`: a list of jobs to treat. Each job contains four relevant fields for the actual computation:
   (a) `posX`,`posY`: the coordinates of the current position in the assembly which `placeTile` should attempt to fill with a tile

(b) `tileset`: the current tileset (which is a vector of integer values which are each the concatenated integer values representing the properties of a tile type)
(c) `assembly`: the current assembly (which is a two-dimensional vector storing the index of the tile type, in the tileset, which is located at each pair of (x, y) coordinates

Note that `placeTile` also makes use of the globally defined two-dimensional vector `pattern` which, at each location representing a pair of (x, y) coordinates, defines one of two colors (i.e. 0 (black) or 1 (white)) for the pattern at that location.

Lemma 13 *For any list of jobs* $\mathtt{j}_0$ *and any function* `save`*,* `placeTile share save` $\mathtt{j}_0$ *returns the list of all subjobs of jobs of* $\mathtt{j}_0$ *that have not been explored, and all results that have been found during the exploration of the explored subjobs of* $\mathtt{j}_0$*.*

Moreover, all its calls to `save` *are of the form* `save j r`*, where* `j` *is the list of all subjobs of* $\mathtt{j}_0$ *that have not been completely explored, and* `r` *is the list of all results that have been found in the exploration of all other subjobs of* $\mathtt{j}_0$*.*

Proof We will prove, by induction on the number of subjobs of $\mathtt{j}_0$, that for all values of `r` and `j`, `placeTile share save r j` is the concatenation of `r` with all the results found in the exploration, and all the subjobs of jobs of `j` that have not been explored.

Moreover, we will prove the following invariant:

Invariant 3 *The recursive calls of* `placeTile` *are all such that the* `results` *and* `jobs` *arguments verify the condition that* `jobs` *is the list of all subjobs of the initial job list that have not been explored and contains no results, and* `results` *is the list of all results that have been found during the exploration of all other subjobs of the initial job list.*

```
let pos a b = (a lsl 16) lor b

let counter =ref 0
let rec placeTile share save results js =
  match js with
      [] →
```

The first case is when the list of jobs to explore is empty. In this case, we simply return the list of found results, and the claim holds.

```
        (results, [])
    | j :: s → (
```

The following expression is the only place where `placeTile` calls `save`. Although it is not purely functional, the arguments `results` and `js` to `save` are non-mutable, and hence not modified by this call. Moreover, by invariant 3 on the recursive calls of `placeTile`, our claim on the calls to `save` clearly holds.

```
if !counter ≥ 10000 then (
  save results js;
  counter := 0;
) else incr counter;
```

First, the variable `inS` is the glue to the south of the location (x, y) to be tiled (i.e. its south input). If the location to the south is outside of the pattern or if there is no tile there, then the glue value of `mgl` is used. In an analogous manner, the variable `inW` is the value of the input glue to the west.

```
let inS =if j.posY ≤ 0 then mgl else
    try
      north (
        IntMap.find
          (IntMap.find (pos j.posX (j.posY − 1)) j.assembly)
          j.tileset
      )
    with
        Not_found → mgl
in
let inW =if j.posX ≤ 0 then mgl else
    try
      east (
        IntMap.find
          (IntMap.find (pos (j.posX − 1) j.posY) j.assembly)
          j.tileset
      )
    with
        Not_found → mgl
in
```

If `inS` (respectively `inW`) is `mgl`, and we are not on the south (respectively west) border, then there is no more tile we can add on the current row (respectively column). Therefore, we must start a new column (respectively a new row). Remark that since we keep alternating between adding rows and columns, we maintain the following invariant: `posX` ≥ `posY` if and only if we are adding a new row. This is what the following code does. Invariant 3, on `placeTile`'s recursive calls, is clearly preserved by all the calls in this portion of the code.

```
if inS = mgl ∧ j.posY > 0 then
  if j.posX < Array.length pattern.(0) then
    placeTile share save results ({ j with posY = 0 } :: s)
  else
    if j.posY + 1 ≥ Array.length pattern then
      placeTile share save (j :: results) s
    else
      placeTile share save results ({ j with posX = 0; posY =
j.posY + 1 } :: s)
```

```
      else
        if inW = mgl ∧ j.posX > 0 then
          if j.posY < Array.length pattern then
            placeTile share save results ({ j with posX = 0 } :: s)
          else
            if j.posX + 1 ≥ Array.length pattern.(0) then
              placeTile share save (j :: results) s
            else
              placeTile share save results ({ j with posY = 0; posX =
j.posX + 1 } :: s)
        else
```

Else, both the south and west glues are defined, or we are at the beginning of a row or a column. Hence, there are two possible cases: either there is already a tile with matching south and west glues, or there is none. In the first case, we have no choice but to place that tile at the current position, and move on to the next position, which is done when `possible` ≥ 0:

```
          let nextX, nextY =
            if j.posY > j.posX then (j.posX+1, j.posY) else (j.posX, j.posY+
1)
          and col = pattern.(j.posY).(j.posX)
          and key = ((inS lsl wgl) lor inW) in
          let possible =
            IntMap.fold (fun k a x →
              if a lsr (3 × wgl) = key then k else x
            ) j.tileset (−1)
          in
          if possible ≥ 0 then
            let tile = IntMap.find possible j.tileset in
            if color tile = col ∨ color tile = mgl then
              placeTile share save results
                ({ posX = nextX; posY = nextY;
                   tileset = IntMap.add possible (withC tile col) j.tileset;
                   assembly = IntMap.add (pos j.posX j.posY) possible j.assembly }
                 :: s)
            else (
              placeTile share save results s
            )
          else
```

Or there is no matching tile, in which case we simply try all tiles that can be placed at the current position, which fall in either of two cases:

- tiles whose color matches the pattern's color at the current position.
- tiles that have not yet been used, i.e. whose color is not yet defined. We only need to consider one of them, because we do not consider solutions

that are equivalent by renaming. This is why we use the `tried_blank` parameter of `IntMap.fold` below.

In both cases, we merge these tiles' west and south glues with `inW` and `inS`, respectively: according to Lemma 12, this means that we adjust the tileset so that the chosen tile can be placed without mismatches at the current position.

```
let _, next_jobs =
  IntMap.fold (fun k t (tried_blank, jobs) →
    if color t = col ∨ (color t = mgl ∧ ¬ tried_blank) then (
      let merged = merge inS (south t) inW (west t) k col j.tileset in
      if isDirected merged then
        (tried_blank ∨ color t = mgl,
         { posX = nextX; posY = nextY;
           tileset = merged;
           assembly = IntMap.add (pos j.posX j.posY) k j.assembly } ::
jobs)
      else (
        (tried_blank ∨ color t = mgl, jobs)
      )
    ) else
      (tried_blank, jobs)
  ) j.tileset (false,s)
in
```

Finally, the following recursive call to `placeTile` maintains the invariant 3: Indeed, `results` is the same as in the initial call, and `next_jobs` now contains `js`, along with all subjobs of `j` (up to renaming of unused tiles).

```
        placeTile share save results next_jobs
)
```

□

The last part of this module uses the client framework proven in Section 5.5 to call the `placeTile` function.

```
let _ =
  Pats.client
    { server = (Unix.gethostbyname "pats.lif.univ-mrs.fr").Unix.h_addr_list.(0);
      port = 5129;
      key = key }
    (fun save j →
      placeTile true save [] [j]
    )
```

5.7 Proof of Lemma 1

We can finally combine all the results of Sect. 5.2 to get our Lemma:

Lemma 1 *If the RSA signatures of all messages used when checking the proof were not counterfeit, then the gadget pattern G, shown in Fig. 2, can only be self-assembled with 13 tile types if a tile set is used which is isomorphic to T.*

Proof The result follows from the combination of Lemmas 7, 10 and 13. □

Acknowledgements We thank Manuel Bertrand for his infinite patience and helpful assistance with setting up the server and helping debug our network and system problems, and Cécile Barbier, Eric Fede and Kai Poutrain for their assistance with software setup.

References

1. Allender E, Koucký M (2010) Amplifying lower bounds by means of self-reducibility. J ACM 57(3):14:1–14:36, DOI 10.1145/1706591.1706594
2. Appel K, Haken W (1977) Every planar map is four colorable. Part I. discharging. Illinois J Math 21:429–490
3. Appel K, Haken W (1977) Every planar map is four colorable. Part II. reducibility. Illinois J Math 21:491–567
4. Barish R, Rothemund PWK, Winfree E (2005) Two computational primitives for algorithmic self-assembly: Copying and counting. Nano Lett 5(12):2586–2592
5. Chow TY (2011) Almost-natural proofs. J Comput Syst Sci 77(4):728–737, DOI 10.1016/j.jcss.2010.06.017
6. Cook M, Rothemund PWK, Winfree E (2004) Self-assembled circuit patterns. In: Proc. 9th International Meeting on DNA Based Computers (DNA9), Springer, LNCS 2943, pp 91–107
7. Czeizler E, Popa A (2013) Synthesizing minimal tile sets for complex patterns in the framework of patterned DNA self-assembly. Theor Comput Sci 499:23–37
8. Demaine ED, Demaine ML, Fekete SP, Patitz MJ, Schweller RT, Winslow A, Woods D (2012) One tile to rule them all: simulating any Turing machine, tile assembly system, or tiling system with a single puzzle piece. Tech. rep., arxiv preprint: `arXiv:1212.4756`
9. Demaine ED, Patitz MJ, Rogers TA, Schweller RT, Summers SM, Woods D (2013) The two-handed tile assembly model is not intrinsically universal. In: Proc. 40th International Colloquium on Automata, Languages and Programming (ICALP2013), Springer, Riga, Latvia, LNCS, vol 7965, pp 400–412, arxiv preprint: `arXiv:1306.6710`
10. Doty D, Lutz JH, Patitz MJ, Summers SM, Woods D (2009) Intrinsic universality in self-assembly. In: Proc. 27th International Symposium on Theoretical Aspects of Computer Science (STACS2009), pp 275–286, arxiv preprint: `arXiv:1001.0208`

11. Doty D, Lutz JH, Patitz MJ, Schweller RT, Summers SM, Woods D (2012) The tile assembly model is intrinsically universal. In: Proc. 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS2012), pp 439–446, arxiv preprint: arXiv:1111.3097
12. Fujibayashi K, Hariadi R, Park SH, Winfree E, Murata S (2007) Toward reliable algorithmic self-assembly of DNA tiles: A fixed-width cellular automaton pattern. Nano Letters 8(7):1791–1797
13. Garey MR, Johnson DS (1979) Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company
14. Gonthier G (2008) Formal proof – the four-color theorem. Not Am Math Soc 55(11):1382–1393
15. Göös M, Lempiäinen T, Czeizler E, Orponen P (2014) Search methods for tile sets in patterned DNA self-assembly. J Comput Syst Sci 80:297–319
16. Hales TC (2000) Cannonballs and honeycombs. Not Am Math Soc 47(4):440–449
17. Helfgott HA (2013) The ternary Goldbach conjecture is true arXiv/1312.7748
18. Johnsen A, Kao MY, Seki S (2013) Computing minimum tile sets to self-assemble color patterns. In: Proc. 24th International Symposium on Algorithms and Computation (ISAAC 2013), Springer, LNCS 8283, pp 699–710
19. Johnsen A, Kao MY, Seki S (2015) A manually-checkable proof for the NP-hardness of 11-color pattern self-assembly tileset synthesis. Journal of Combinatorial Optimization In print. arXiv preprint: arXiv:1409.1619
20. Kari L, Kopecki S, Meunier PE, Patitz MJ, Seki S (2015) Binary pattern tile set synthesis is NP-hard. In: Proc. 42nd International Colloquium on Automata, Languages, and Programming (ICALP2015), Springer, LNCS, vol 9134, pp 1022–1034, arxiv preprint: arXiv:1404.0967
21. Kari L, Kopecki S, Seki S (2015) 3-color bounded patterned self-assembly. Nat Comp 14(2):279–292
22. Konev B, Lisitsa A (2014) A SAT attack on the Erdös discrepancy conjecture. arXiv: 1402.2184
23. Lathrop JI, Lutz JH, Patitz MJ, Summers SM (2011) Computability and complexity in self-assembly. Theory Comput Syst 48(3):617–647
24. Lund K, Manzo AT, Dabby N, Micholotti N, Johnson-Buck A, Nangreave J, Taylor S, Pei R, Stojanovic MN, Walter NG, Winfree E, Yan H (2010) Molecular robots guided by prescriptive landscapes. Nature 465:206–210
25. Ma X, Lombardi F (2008) Synthesis of tile sets for DNA self-assembly. IEEE T Comput Aid D 27(5):963–967
26. Marchal C (2011) Study of the Kepler's conjecture: The problem of the closest packing. Math Z 267(3-4):737–765
27. Meunier PE, Patitz MJ, Summers SM, Theyssier G, Winslow A, Woods D (2014) Intrinsic universality in tile self-assembly requires cooperation. In: Proc. 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA2014), pp 752–771, arxiv preprint: arXiv:1304.1679
28. Mulzer W, Rote G (2008) Minimum-weight triangulation is NP-hard. J ACM 55(2):Article No. 11

29. Patitz MJ, Summers SM (2011) Self-assembly of decidable sets. Nat Comp 10(2):853–877
30. Qian L, Winfree E (2011) Scaling up digital circuit computation with DNA strand displacement cascades. Science 332(6034):1196
31. Qian L, Winfree E, Bruck J (2011) Neural network computation with DNA strand displacement cascades. Nature 475(7356):368–372
32. Razborov AA, Rudich S (1994) Natural proofs. In: Proc. 26th Annual ACM Symposium on Theory of Computing (STOC1994), ACM, New York, NY, USA, pp 204–213, DOI 10.1145/195058.195134
33. Robertson N, Sanders DP, Seymour P, Thomas R (1996) A new proof of the four-colour theorem. Electron Res Announc AMS 2(1):17–25
34. Rothemund PW (2006) Folding DNA to create nanoscale shapes and patterns. Nature 440(7082):297–302
35. Rothemund PW, Papadakis N, Winfree E (2004) Algorithmic self-assembly of DNA Sierpinski triangles. PLoS Biol 2(12):2041–2053
36. Rudich S (1997) Super-bits, demi-bits, and NP/qpoly-natural proofs. J Comput Syst Sci 55:204–213
37. Seelig G, Soloveichik D, Zhang DY, Winfree E (2006) Enzyme-free nucleic acid logic circuits. Science 314(5805):1585–1588
38. Seeman NC (1982) Nucleic-acid junctions and lattices. J Theor Biol 99:237–247
39. Seki S (2013) Combinatorial optimization in pattern assembly (extended abstract). In: Proc. 12th International Conference on Unconventional Computation and Natural Computation (UCNC 2013), Springer, LNCS 7956, pp 220–231
40. Sterling A (????) https://nanoexplanations.wordpress.com/2011/08/13/dna-self-assembly-of-multicolored-rectangles/
41. Szekeres G, Peters L (2006) Computer solution to the 17-point Erdös-Szekeres problem. The ANZIAM Journal 48:151–164
42. Tuckerman B (1971) The 24th Mersenne prime. Proc Nat Acad Sci USA 68:2319–2320
43. Wang H (1961) Proving theorems by pattern recognition – II. AT&T Tech J XL(1):1–41
44. Wei B, Dai M, Yin P (2012) Complex shapes self-assembled from single-stranded DNA tiles. Nature 485(7400):623–626
45. Winfree E (1998) Algorithmic self-assembly of DNA. PhD thesis, California Institute of Technology
46. Winfree E, Liu F, Wenzler LA, Seeman NC (1998) Design and self-assembly of two-dimensional DNA crystals. Nature 394(6693):539–44
47. Woods D (2013) Intrinsic universality and the computational power of self-assembly Arxiv preprint: arXiv:1309.1265
48. Yan H, Park SH, Finkelson G, Reif JH, LaBean TH (2003) DNA-templated self-assembly of protein arrays and highly conductive nanowires. Science 301:1882–1884
49. Yurke B, Turberfield AJ, Mills AP, Simmel FC, Neumann JL (2000) A DNA-fuelled molecular machine made of DNA. Nature 406(6796):605–608

50. Zhang J, Liu Y, Ke Y, Yan H (2006) Periodic square-like gold nanoparticle arrays templated by self-assembled 2D DNA nanogrids on a surface. Nano Letters 6(2):248–251