Theoretical Computer Science 958 (2023) 113848

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Sublinear P system solutions to NP-complete problems [☆]

Michael J. Dinneen, Alec Henderson [*], Radu Nicolescu

School of Computer Science, The University of Auckland, Auckland, New Zealand

A R T I C L E   I N F O

Article history:
Received 2 November 2021
Received in revised form 3 March 2023
Accepted 24 March 2023
Available online 5 April 2023

We dedicate this paper to Cris Calude for his 70th birthday in recognition to his work on unconventional (models of) computing for the last 20+ years [3,2].

Keywords:
cP systems
Membrane computing
Computational complexity
NP-complete

A B S T R A C T

Many membrane systems (e.g. P System), including cP systems (P Systems with compound terms), have been used to solve efficiently many NP-hard problems, often in linear time. However, these solutions have been independent of each other and have not utilised the theory of reductions. This work presents a sublinear solution to k-SAT and demonstrates that k-colouring can be reduced to k-SAT in constant time. This work demonstrates that traditional reductions are efficient in cP systems and that they can sometimes produce more efficient solutions than the previous problem-specific solutions.

1. Introduction

The question of whether P equals NP is unquestionably the most important unsolved problem in computational complexity theory. The problem has been studied extensively, with many practical problems found to be NP-complete. However, the currently best-known general solutions to NP-complete problems take prohibitively large amounts of time for large instances.

P systems are a parallel and distributed model of computing, first proposed by Gheorghe Paun in [16]. P systems are abstract models of membrane systems, with many variants being proposed such as: P systems with active membranes [17], spiking neural P systems [9], tissue P systems [12], and P systems with compound terms (cP systems) [14]. These systems have been found to have efficient solutions to hard problems. However, as far as we know, these efficient solutions are still in theory and have not yet been practically realised.

cP systems have been used to solve well-known NP-hard problems efficiently such as: the Hamiltonian path, travelling salesman [6], 3-colouring [5], and subset sum [10]. However, these solutions have been made specifically for each problem, without utilising the theory of reductions. In this work, we propose: to the best of our knowledge, (1) the most time (number of cP steps) efficient solution to k-SAT using P and cP systems, and (2) the most time-efficient solution to the 3-colouring problem using a *cP system reduction* of it to our k-SAT solution.

☆ This article belongs to Section C: Theory of natural computing, Edited by Lila Kari.
* Corresponding author.
E-mail addresses: m.dinneen@auckland.ac.nz (M.J. Dinneen), ahen386@aucklanduni.ac.nz (A. Henderson), r.nicolescu@auckland.ac.nz (R. Nicolescu).

https://doi.org/10.1016/j.tcs.2023.113848
0304-3975/

As discussed in [19], the Cook–Levin theorem states that there exists a polynomial-time reduction from every problem in NP to SAT. For many problems, the reduction used in the proof is not the most efficient. Utilising this work, we demonstrate how k-colouring can reduce to k-SAT in constant time using cP systems. This work should allow for more efficient solutions to already solved problems using reductions and hopefully inspire more work to focus on general issues rather than specific problem instances.

2. Background

In this section, we describe NP-completeness with emphasis on the Boolean satisfiability (SAT) problem. We discuss polynomial-time reductions and give a reduction from k-colouring to k-SAT. We finally introduce cP systems as a membrane computing model and give example rules.

2.1. NP-completeness

NP-complete languages have been studied for decades, and knowing whether the associated complexity class is within P is one of the most important questions in theoretical computer science. A decision problem is NP-complete if and only if it is NP-hard and it belongs to the class NP. NP-hardness means that we can reduce in polynomial-time any language in NP to it. Many of these problems have significant practical importance. There are hundreds, if not thousands, of problems that have been found to be NP-complete. Of course, NP-completeness deals with decision problems; however, all NP-complete languages being self-reducible means we do not need to study optimisation versions as much [1].

2.1.1. SAT

The Boolean satisfiability problem (SAT) is one of the most famous NP-complete problems and also the first problem shown to be NP-complete [4]: given a Boolean formula, does there exist a satisfying truth assignment? Typically the problem considers formulas in conjunctive normal form (CNF). A Boolean formula is in CNF if it is expressed as a *conjunction* ($\wedge$) of clauses. A *clause* is a *disjunction* ($\vee$) of literals. A *literal* is a variable or its negation (here indicated by overbars). For example, the following Boolean formula is in CNF:

$$(x_1 \vee x_2) \wedge (\bar{x_1} \vee \bar{x_2})$$

The k-SAT problem is a restricted version of SAT, where each clause contains at most k literals. This restricted version is also NP-complete for any $k \geq 3$.

2.1.2. Polynomial-time reductions

As defined in [18], given two languages $A \subseteq \Sigma^*$, $B \subseteq \Psi^*$, A is polynomial-time mapping reducible (also known as Karp reducible) to a language B ($A \leq_p B$) if a polynomially computable function $f : \Sigma^* \to \Psi^*$ exists where for every w:

$$w \in A \iff f(w) \in B$$

The function f is called the polynomial-time reduction. The well-known proof by Cook [4] shows how all languages in NP have a polynomial-time reduction to SAT. In this section, we describe the reduction from 3-colouring to SAT [19].

2.1.3. Useful Boolean formulas

As discussed in [19], we can make some useful formulas in CNF, which simplify the reductions to SAT. *at_most_one* is a formula which defines the property that only one literal of the arguments is **true**:

$$\mathit{at\_most\_one}(l_1, l_2, \ldots, l_n) = \bigwedge_{1 \leq i < j \leq n} (\bar{l_i} \vee \bar{l_j})$$

Similarly we can define *at_least_one* meaning at least one variable is **true**:

$$\mathit{at\_least\_one}(l_1, l_2, \ldots, l_n) = (l_1 \vee l_2 \vee \cdots \vee l_n)$$

Combining these, we can also define *exactly_one* where exactly one of the variables will be **true**:

$$\mathit{exactly\_one}(l_1, l_2, \ldots, l_n) = \mathit{at\_most\_one}(l_1, l_2, \ldots, l_n) \wedge \mathit{at\_least\_one}(l_1, l_2, \ldots, l_n)$$

The formula given for *at_most_one* is not the most efficient and can be implemented in $O(n)$ rather than the $O(n^2)$ version we defined, as discussed in [19]. This more efficient implementation does, however, introduce $n - 2$ more variables. Throughout this paper, we shall assume this inefficient encoding.

Table 1
BNF grammar for cP top-cells as presented in [8].

```
<top−cell> ::= <state> <term> ...
<state> ::= <atom>
<term> ::= <atom> | <sub−cell>
<sub−cell> ::= <compound−term> ...
<compound−term> ::= <functor> <args> ...
<functor> ::= <atom>
<args> ::= ‘(’ <term> ... ‘)’
```

Table 2
Restricted BNF grammar for cP rules, α being the application mode, $\alpha \in \{1, +\}$.

```
<rule> ::= <lhs> →a <rhs> <promoters>
<lhs> ::= <state> <vterm> ...
<rhs> ::= <state> <vterm> ...
<promoters> ::= (‘|’ <vterm>) ...
<vterm> ::= <variable> | <atom> | <compound−vterm>
<compound−vterm> ::= <functor> <vargs> ...
<vargs> ::=  ‘(’<vterm> ...‘)’
```

2.1.4. k-colouring

As defined in [19], given an undirected graph G with vertices V and edges E, G is *k-colourable* if there exists a function:

$$f : V \to \{1, 2, \dots, k\} \text{ such that for all } \{u, v\} \in E,\ f(u) \neq f(v).$$

Here we show the reduction from an instance of k-colouring to SAT given in [19]. The set of variables is denoted X, formula as F, and a set K as $\{1, 2, \dots, k\}$.

$$X = \{x_{v,i} : v \in V, i \in K\}$$
$$F = \bigwedge_{v\in V} exactly\_one(x_{v,i} : i \in K) \wedge \bigwedge_{\{u,v\}\in E} \bigwedge_{i\in K} at\_most\_one(x_{u,i}, x_{v,i}) \tag{1}$$

Each vertex in the graph is represented in the formula by k variables, where each variable in the formula represents a vertex assigned to the colour i.

The first part of the formula ($\bigwedge_{v\in V} exactly\_one(x_{v,i} : i \in K)$) represents the requirement that a vertex must take exactly one colour. This will create $|V|\binom{k}{2}+|V|k$ clauses, i.e. $O(|V|k^2)$.

The second part of the formula ($\bigwedge_{\{u,v\}\in E} \bigwedge_{i\in K}(\bar{x}_{u,i} \vee \bar{x}_{v,i})$) ensures that each pair of vertices connected by an edge will have a different colour. This will create $O(|E|k)$ clauses, with each clause containing two variables.

Theorem 1. *The k-colourable clause set is linear in the input size.*

Based on the previous analysis, we know that we create $O(|V|k^2)$ clauses for the first part of the formula. We also know that we create $O(|E|k)$ clauses for the second part, each being two variable lengths. Therefore the entire formula is $O(|V|k^2 + |E|k)$ characters. Due to k being a fixed constant, we know the length will be $O(|V| + |E|)$, which is linear in the input size.

2.2. cP systems

cP systems are a parallel and distributed model of computation, which utilises high-level rewriting rules to compute efficient solutions to problems. In this section, we shall briefly discuss the grammar and rule execution, with a focus on the types of rules used in this paper and not on the general framework of cP systems. We shall highlight how the rules work via examples and direct the reader to [14] for a more in-depth explanation on cP systems.

A cP system consists of a top-level cell (can be many, but we do not consider that in this paper) and subcells following the grammar presented in Table 1, where the notation ‘...’ represents 0 or more repetitions of the previous symbol. The subcells do not contain rules and are practically just a data storage facility.

The system evolves based on high-level rewriting rules obeying the grammar presented in Table 2. Before a rule can apply, it must match all conditions on the left-hand side and right-hand side promoters by way of multiset unification. *vterm* arguments require a complete match. There are two rule application modes: exactly once ($\rightarrow_1$), and maximally parallel ($\rightarrow_+$). Exactly once will apply a rule for one matching, whereas a maximally parallel rule will apply it as many times as possible, all in the same step.

Rules are applied in weak priority order, with rules considered in a 'top down' order. Once an applicable rule has been found, this commits the next state, with subsequent rules committing to different states disabled. Rules going to the same state as the applicable rule, which can also be applied, will be applied in the same step. By convention, we denote the ith state by s_i and follow this convention throughout the remainder of the text.

Numbers in the system are represented using unary. Whereby convention, we denote the unary symbol as *1* and 1^x as x. As it is simple to transfer between the representations throughout this paper, we shall use the numbers rather than the low-level unary representation, including for 0 (λ). For example:

$$3 = 111$$
$$0 = \lambda$$

We give two examples to clarify how cP systems are defined and used.

Example 1. The problem of incrementing all numbers with functor n by 1 can be done in one step using the rule:

$$s_1 \; n(X) \quad \rightarrow_+ \quad s_2 \; n(1X) \qquad (1)$$

This rule runs in maximally parallel mode, meaning all of the instances of n would be incremented. Whereas if we used the rule:

$$s_1 \; n(X) \quad \rightarrow_1 \quad s_2 \; n(1X) \qquad (1)$$

Exactly one of the instances would be incremented. If there were multiple instances the system would non-deterministically choose one of them.

Example 2. Given a system with numbers x and y, multiplication ($z = xy$) is achieved using the following rules:

$$\begin{array}{lll l} s_0 & \rightarrow_+ & s_1 \; z(0) & (1) \\ s_1 \; y(0) & \rightarrow_+ & s_2 & (2) \\ s_1 \; x(X) \; y(1Y) \; z(Z) & \rightarrow_+ & s_1 \; z(ZX) \; y(Y) & (3) \end{array}$$

Rule 3 is a loop that will subtract one from y and add the value of x to z. Rule 3 will be the only rule able to be applied until y reaches zero. Once y is zero rule 2 will be applied. Because they have different states, rule 3 would not be applied in parallel (it would not be applied anyway as $y(1Y)$ cannot match $y(0)$).

3. Ruleset for k-SAT

Here we assume k-SAT to be a formula in CNF with variables $x_0, x_1, \ldots, x_{n-1}$ where each clause contains at most k variables. To solve k-SAT in $O(\sqrt{n})$ time, we break it up into three steps: generating assignment templates, generating the assignments, and finally evaluating the entire formula. Fig. 1 shows a state diagram of the entire system broken down into three main parts.

3.1. Initial configuration

During the execution of the algorithm subcell $m()$ is used to determine when loops have finished; initially set to $m(1)$. Subcell $j()$ is used to store the *branch number* of allocations of variables where the *branch number* is an index of the paths from the root to leaf starting at 0 for the left-most leaf (see Fig. 5b); initially we have $j(0)$ $j(1)$ $j(2)$ $j(3)$ because we assume that we have already allocated x_0 and have the next level of branches ready to assign.(no matter what the value of n we assume that we have only allocated the first variable).

Another way of looking at branch numbers is a bijection between integers and allocations. Algorithms to go between these representations are given in the appendix.

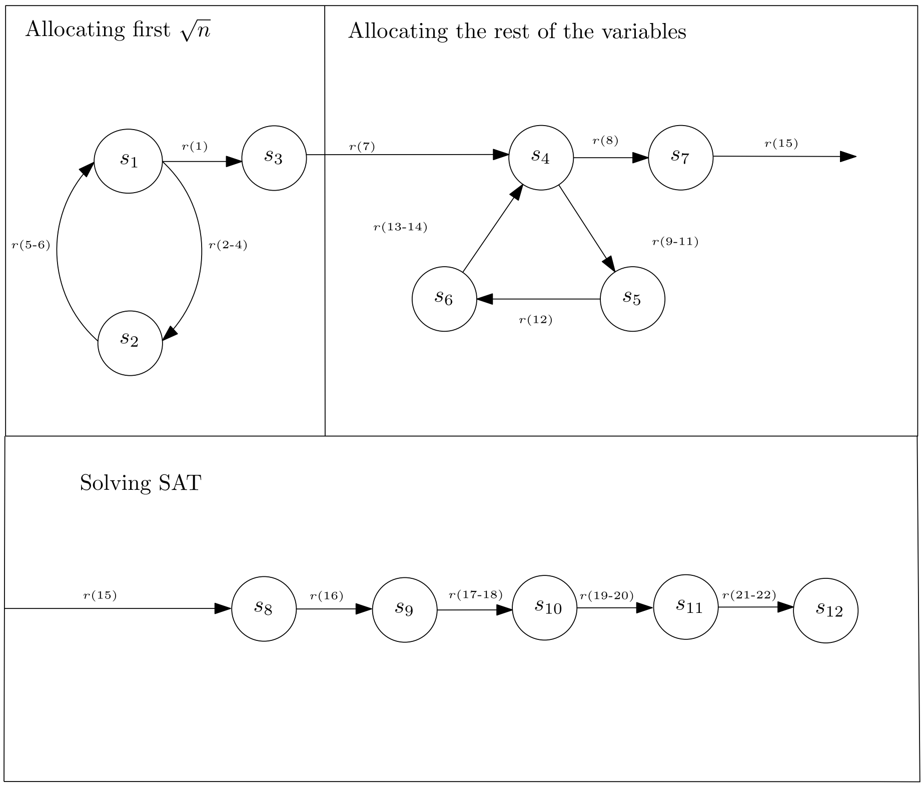


Fig. 1. State diagram broken up into the three parts of the algorithm. Here we have denoted the rules that change states by $r(i)$.

Subcell $a(i)(v)(j)$ states that variable x_i has been assigned the value v, and branch number j. Subcells a initially have x_0 assigned $a(0)(0)(0)$ or $a(0)(1)(1)$.

As described in [8], we can simplify the rules to evaluate a variable using lookup tables as seen in Table 7. The lookup table y has three parts: the assigned value of the variable; the 'sign' of the variable (whether negated or not) in a clause; and the value when you apply this sign to the value of the assigned variable.

The subcell $k()$ contains the value k for k-SAT. The subcell ρ contains $\lceil\sqrt{n}\rceil$ (ceiling rounded) where n is the number of variables. The formula that is being tested is encoded as subcells $c(x(i)(s)\ldots)$ where i denotes which x_i is being referred to and s whether it is negated $(-)$ in the clause.

3.2. Allocating first $\sqrt{n}$ variables

First, we allocate the first $\sqrt{n}$ variables, which are then used as a lookup table. A sequential programming approach to creating these allocations can be seen in Fig. 2. The outer loop is used to reference the next variable being assigned and the first inner loop creates two new allocations from the previous ones. The second inner loop allocates the next variable for these newly created allocations a 0 if the branch is even, and a 1 if odd.

Our cP system closely models that of the sequential algorithm presented in Fig. 2 with the ruleset presented in Table 3. Rules 1 and 6 form the outer loop, with rule 6 being the increment and rule 1 being the termination condition. Rule 2 creates the copies and changes their branch numbers. Rules 3 and 4 add the next variable to the allocations. Rule 5 updates the branch numbers ready for the next iteration of the loop. The outer loop formed by rules 1 and 6 runs $\sqrt{n}$ times. The inner loop runs in parallel for all allocations at once. Rules 2-4 run in parallel, taking 1 step total for each loop. Rules 5 and 6 also run in parallel making the total running time $2\sqrt{n}+1$ alternatively $O(\sqrt{n})$.

3.3. Allocating all other variables

To allocate the rest of the variables, we use the templates that were previously created for the first $\sqrt{n}$ variables. Using the templates, we loop $\sqrt{n}$ times, where on each loop we do a Cartesian product between the previously allocated variables and the template (the templates variables get incremented by $\sqrt{n}$ before each Cartesian product). Alternatively, this operation can be viewed as taking the allocation tree in Table 8, copying it and placing the tree at all of the leaves, as shown in Fig. 3.

```
$a \leftarrow \{(0,0,0),(0,1,1)\}$
$j \leftarrow \{0,1,2,3\}$
for $m \leftarrow 1$ to $\sqrt{n}$ do
    $p \leftarrow \{\}$
    for $(i,v,j)$ in $a$ do
        $p \leftarrow p \cup (i,v,j*2) \cup (i,v,j*2+1)$
    $t \leftarrow \{\}$
    for $z$ in $j$ do
        if $z\%2 = 0$
            $p \leftarrow p \cup (m,0,z)$
        else
            $p \leftarrow p \cup (m,1,z)$
        $t \leftarrow t \cup 2*z \cup 2*z+1$
    $a \leftarrow p$
    $j \leftarrow t$
```

Fig. 2. Sequential algorithm for creating first $\sqrt{n}$ allocations.

Table 3
cP rules to allocate first $\sqrt{n}$ variables.

$s_1\ m(I)$	$\rightarrow_1$	$s_3\ m(I)$ $\mid l(I)$	(1)
$s_1\ a(X)(Y)(Z)$	$\rightarrow_+$	$s_2\ a(X)(Y)(ZZ)$ $a(X)(Y)(ZZ1)$	(2)
$s_1\ j(ZZ)$	$\rightarrow_+$	$s_2\ j(ZZ)\ a(Y)(0)(ZZ)$ $\mid m(Y)$	(3)
$s_1\ j(ZZ1)$	$\rightarrow_+$	$s_2\ j(ZZ1)\ a(Y)(1)(ZZ1)$ $\mid m(Y)$	(4)
$s_2\ j(Z)$	$\rightarrow_+$	$s_1\ j(ZZ)$ $j(ZZ1)$	(5)
$s_2\ m(I)$	$\rightarrow_1$	$s_1\ m(I1)$	(6)

Table 4
Sequential algorithm for creating the rest of the allocations.

```
$a \leftarrow \{(0,0,0),\dots\}$
$b \leftarrow$ map $a$ by $(i,v,j) \rightarrow (i+\sqrt{n},v,j)$
$p \leftarrow \{\}$
for $m \leftarrow \sqrt{n}$ to $n$ step $\sqrt{n}$ do
    $d \leftarrow \{\}$
    for $(i,v,j)$ in $a$ do
        for $(y,q,x)$ in $b$ do
            if $i+m=y$ then
                $d \leftarrow d \cup (i,v,(j,x))$
                $d \leftarrow d \cup (y,v,(j,x))$
            for z in p do
                if $i+z=y$ then
                    $d \rightarrow d \cup (i,v,(j,x))$
    $p \leftarrow p \cup m$
    $a \leftarrow d$
    $b \leftarrow$ map $b$ by $(i,v,j) \rightarrow (i+\sqrt{n},v,j)$
```

A sequential version of this algorithm can be seen in Table 4. First, we make a copy of the template a with all variables incremented by $\sqrt{n}$. Then we do an outer loop from $\sqrt{n}$ to n incrementing by $\sqrt{n}$. Inside this loop, a Cartesian product is made looping over each allocation in b and in a. The branch number for the combined allocation is denoted recursively as $\alpha(j)(i)$, with j being the branch number from a, and i being the branch number from b. For example, for 9 variables, one of the branch numbers created is $\alpha(\alpha(0)(1))(2)$.

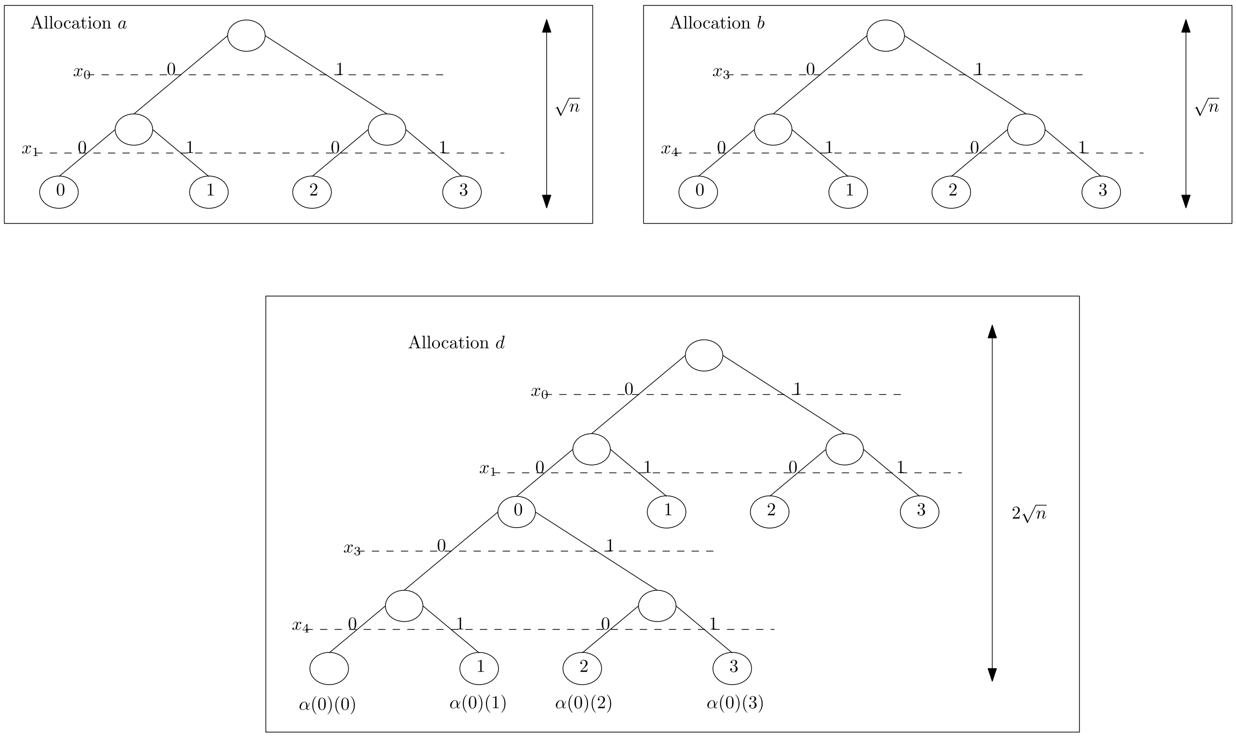


Fig. 3. Diagrams representing the Cartesian product showing for $\alpha = 0$ and $n = 4$.

The rules to allocate the remaining variables are in Table 5. The state diagram giving the state transitions of allocating the variables is shown in Fig. 1.

Rule 7 acts as allocating the original b value. Rules 8 and 11 form the outer loop presented in the sequential algorithm, with rule 8 the termination condition and rule 11 the increment. Rules 9 and 10 apply the Cartesian product. Rule 13 increments b (the last line of the sequential algorithm), and rules 12 and 14 reassign a (line 15 of the sequential algorithm).

The rules 9-14 form a loop, which runs $\sqrt{n}$ times, with rules 9-11 running in parallel as well as rules 13 and 14. The loop takes $3\sqrt{n}$ steps and rules 7 and 8 each take one step, making the running time for the rules presented in Table 5 $3\sqrt{n}+2$ steps.

3.4. *Solving SAT*

The rules discussed previously are just a way of allocating all of the variables. For each complete allocation, the formula is evaluated. Finally, it is checked if any satisfy the formula. A sequential algorithm describing the steps taken by our cP system can be found in Fig. 4. A state diagram of this final part of the algorithm is shown in Fig. 1.

The ruleset presented in Table 6 is the cP system equivalent of the algorithm presented in Fig. 4. Rules 15 and 16 are used to copy the formula for each of the different allocations. Rule 17 checks if any of the allocated variables makes the clause true if none exists, rule 18 sets the clause to false (they apply the **or** operation $\vee$). Rules 19 and 20 apply the **and** operation ($\wedge$) between the clauses. Rules 21 and 22 determine if a satisfying truth assignment exists, outputting $r(1)$ if one existed, and $r(0)$ otherwise.

Rules 15 and 16 run once and are independent of each other (2 steps). The pairs of rules (17, 18) (19, 20), and (21, 22) each run once, with each pair taking 1 step (total is 3 steps). In summary, the running time of the ruleset presented in Table 6 is 5 steps.

Theorem 2. *k-SAT is solvable in $O(\sqrt{n})$*

The rulesets presented in Tables 3, 5 and 6 solve k-SAT with the running times being $\sqrt{n}+1$, $3\sqrt{n}+2$ and 5 making the total time $4\sqrt{n}+8$, or $O(\sqrt{n})$.

Table 5
Rules to create allocations.

$s_3\ a(X)(Y)(Z)$	$\rightarrow_+$	$s_4\ a(X)(Y)(Z)$	(7)
		$\quad b(XQ)(Y)(Z)$	
		$\mid l(Q)$	
		$\mid n(XQI)$	
$s_4\ m(QX)$	$\rightarrow_1$	$s_7\ m(1)$	(8)
		$\mid n(Q)$	
s_4	$\rightarrow_+$	$s_5\ d(X)(Y)(\alpha(Z)(W))$	(9)
		$\quad d(XP)(V)(\alpha(Z)(W))$	
		$\mid a(X)(Y)(Z)$	
		$\mid b(XP)(V)(W)$	
		$\mid m(P)$	
s_4	$\rightarrow_+$	$s_5\ d(X)(Y)(\alpha(Z)(W))$	(10)
		$\mid a(X)(Y)(Z)$	
		$\mid b(XP)(V)(W)$	
		$\mid p(P)$	
$s_4\ m(I)$	$\rightarrow_1$	$s_5\ p(I)\ m(IQ)$	(11)
		$\mid l(Q)$	
$s_5\ a(X)$	$\rightarrow_+$	s_6	(12)
$s_6\ b(X)(Y)(Z)$	$\rightarrow_+$	$s_4\ b(XQ)(Y)(Z)$	(13)
		$\mid L(Q)$	
		$\mid n(XQI)$	
$s_6\ d(X)$	$\rightarrow_+$	$s_4\ a(X)$	(14)

```
$a \leftarrow \{(0, 0, (\alpha(\dots)(0))), \dots\}$
$c \leftarrow \{((i, s), (j, t), \dots), \dots\}$
// a set of $k$ tuples with each $k$ tuple item being a pair.
$\gamma \leftarrow \{\}$
for $(i, v, j)$ in a do
    $\gamma \leftarrow \gamma \cup j$
$\kappa \leftarrow \{\}$
for $d$ in $c$ do
    for j in $\gamma$ do
        $\kappa \leftarrow \kappa \cup (d, j)$
$t \leftarrow \{\}$
for (d,j) in $\kappa$ do
    $p \leftarrow 0$
    for $(i, s)$ in $d$ do
        for $(x, v, y)$ in $a$ do
            if $x = i$ and $j = y$ and $y(v, s) = 1$ then
                $p \leftarrow 1$
    $t \leftarrow t \cup (p, j)$
$\kappa \leftarrow t$
$f = \{\}$
for $y$ in $\gamma$ do
    $v \leftarrow 1$
    for $(p, j)$ in $\kappa$ do
        if $p = 0$ and $j = y$ then
            $v \leftarrow 0$
    $f \leftarrow f \cup v$
$r \leftarrow 0$
for $v$ in $f$ do
    if $v = 1$ then
        $r \leftarrow 1$
```

Fig. 4. Sequential algorithm for solving SAT given all the allocations. a and c are the allocations and clauses described earlier (allocated here for self containment).

Table 6
Rules to solve using the allocations.

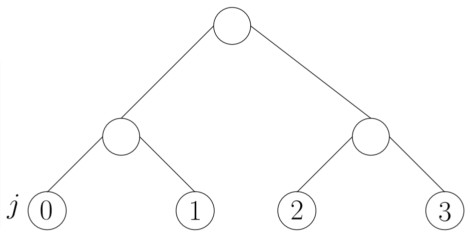
$s_7\ a(0)(Y)(Z)$	$\to_+$	$s_8\ a(0)(Y)(Z)\ \gamma(Z)$	(15)
s_8	$\to_+$	$s_9\ \kappa(X\ j(Y))$ $\mid c(X)$ $\mid \gamma(Y)$	(16)
$s_9\ \kappa(x(I)(S)\ \_\ j(Y))$	$\to_+$	$s_{10}\ \kappa(1)(Y)$ $\mid a(I)(V)(Y)$ $\mid y(V)(S)(1)$	(17)
$s_9\ \kappa(\_\ j(Y))$	$\to_+$	$s_{10}\ \kappa(0)(Y)$	(18)
$s_{10}\ \kappa(0)(Y)\ \gamma(Y)$	$\to_+$	$s_{11}\ f(0)(Y)$	(19)
$s_{10}\ \kappa(1)(Y)\ \gamma(Y)$	$\to_+$	$s_{11}\ f(1)(Y)$	(20)
$s_{11}\ f(1)(\_)\ m(1)$	$\to_+$	$s_{12}\ r(1)$	(21)
$s_{11}\ f(0)(\_)\ m(1)$	$\to_+$	$s_{12}\ r(0)$	(22)

Table 7
Initial state of the subcells that *do not* change, for Formula (2).

Table representation			cP representation
y			
0	+	0	$y(0)(+)(0)$
1	+	1	$y(1)(+)(1)$
0	−	1	$y(0)(-)(1)$
1	−	0	$y(1)(-)(0)$
c			
x_0	x_1		$c(x(0)(+)\ x(1)(+))$
x_2	$\bar{x_3}$		$c(x(2)(+)\ x(3)(-))$
$\bar{x_2}$	x_1		$c(x(2)(-)\ x(1)(+))$
			$k(2)$
			$\rho(2)$

i	x_i	j	cP representation
0	0	0	$a(0)(0)(0)$
0	1	1	$a(0)(1)(1)$
			$m(1)$
			$j(0)\ j(1)\ j(2)\ j(3)$

(a) Table representation.



(b) Graph representation

Fig. 5. cP (a table) and binary tree representation of initial branch numbers.

3.5. High-level example of execution

Here we present an overview of the evaluation of our solution. We do not fully trace the solution but highlight the important steps. We consider the following formula ($n = 4$, $k = 2$, and $\rho = \sqrt{n} = 2$):

$$(x_0 \vee x_1) \wedge (x_2 \vee \bar{x_3}) \wedge (\bar{x_2} \vee x_1). \tag{2}$$

Table 7 contains the fixed subcells, and Fig. 5(a) the subcells that change during the evolution of the system.

The subcells in Fig. 5(a) will change to the subcells in Table 8 after the execution of rules 2-6. For example, rule 7 will create the b subcells displayed in Table 5. Where we note the only difference between the a and b is the first parameter with b subcells have $\sqrt{n}$ added. Rules 9 and 10 will create subcells d which as seen in Table 9 can be viewed as taking a Cartesian product of a and b. The d cells will become the a cells and loop until all allocations have been completed.

Table 8
First step of the algorithm for solving Formula (2).

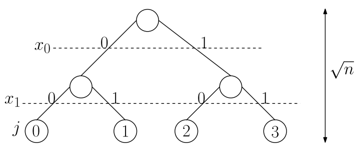
Table representation				Graph representation
i	x_i	j	cP system representation	
0	0	0	$a(0)(0)(0)$	 x_0 0 1 x_1 0 1 0 1 j 0 1 2 3 $\sqrt{n}$
1	0	0	$a(1)(0)(0)$	
0	0	1	$a(0)(0)(1)$	
1	1	1	$a(1)(1)(1)$	
0	1	2	$a(0)(1)(2)$	
1	0	2	$a(1)(0)(2)$	
0	1	3	$a(0)(1)(3)$	
1	1	3	$a(1)(1)(3)$	
			$m(2)$	
			$j(0)\ \ j(1)\ \ j(2)\ \ j(3)$	
			$j(4)\ \ j(5)\ \ j(6)\ \ j(7)$	

Table 9
Creating the next $\sqrt{n}$ variables for Formula (2).

		a	
i	x_i	j	cP representation
0	0	0	$a(0)(0)(0)$
1	0	0	$a(1)(0)(0)$
0	0	1	$a(0)(0)(1)$
1	1	1	$a(1)(1)(1)$
0	1	2	$a(0)(1)(2)$
1	0	2	$a(1)(0)(2)$
0	1	3	$a(0)(1)(3)$
1	1	3	$a(1)(1)(3)$

		b	
i	x_i	j	cP representation
2	0	0	$b(2)(0)(0)$
3	0	0	$b(3)(0)(0)$
2	0	1	$b(2)(0)(1)$
3	1	1	$b(3)(1)(1)$
2	1	2	$b(2)(1)(2)$
3	0	2	$b(3)(0)(2)$
2	1	3	$b(2)(1)(3)$
3	1	3	$b(3)(1)(3)$

		d	
i	x_i	j	cP representation
0	0	$\alpha(0)(0)$	$d(0)(0)(\alpha(0)(0))$
1	0	$\alpha(0)(0)$	$d(1)(0)(\alpha(0)(0))$
2	0	$\alpha(0)(0)$	$d(2)(0)(\alpha(0)(0))$
3	0	$\alpha(0)(0)$	$d(3)(0)(\alpha(0)(0))$

Table 10
Allocating variables for solving Formula (2) where we only list $\alpha(0)$ and $\beta(0)$ for brevity.

Table representation			cP representation
i	x_i	j	
0	0	$\alpha(0)(0)$	$a(0)(0)(\alpha(0)(0))$
1	0	$\alpha(0)(0)$	$a(1)(0)(\alpha(0)(0))$
2	0	$\alpha(0)(0)$	$a(2)(0)(\alpha(0)(0))$
3	0	$\alpha(0)(0)$	$a(3)(0)(\alpha(0)(0))$

Table 11
The clauses for each allocation of Formula (2), where we only list $\alpha(0)(0)$ for brevity.

Table representation			cP representation
$x : i, s$	$x : i, s$	j	
$0, +$	$1, +$	$\alpha(0)\ \beta(0)$	$\kappa(x(0)(+)\ x(1)(+)\ j(\alpha(0)(0)))$
$2, +$	$3, -$	$\alpha(0)\ \beta(0)$	$\kappa(x(0)(+)\ x(1)(-)\ j(\alpha(0)(0)))$
$2, -$	$1, +$	$\alpha(0)\ \beta(0)$	$\kappa(x(0)(-)\ x(1)(+)\ j(\alpha(0)(0)))$

The allocation displayed in Table 10 rule 16 will create a clause for each of the allocations resulting in subcells denoted κ with a branch number as seen in Table 11. Once created, these clauses are evaluated using rules 17 and 18 as shown in Table 12. After the evaluation, the clauses with matching j are combined using an **and** operation $\wedge$ (rules 22 and 23) as shown in Table 13. Finally, the system checks if there is any f subcell containing a 1; if there is, then there exists a satisfying truth assignment.

Table 12
The clause with all variables assigned for Formula (2), where we only list $\alpha(0)(0)$ for brevity.

Table representation		cP representation
v	j	
0	$\alpha(0)(0)$	$\kappa(0)(\alpha(0)(0))$
1	$\alpha(0)(0)$	$\kappa(1)(\alpha(0)(0))$
1	$\alpha(0)(0)$	$\kappa(1)(\alpha(0)(0))$

Table 13
The clause with all variables assigned for Formula (2), where we only list $\alpha(0)(0)$ for brevity.

Table representation		cP representation
v	j	
0	$\alpha(0)(0)$	$f(0)(\alpha(0)(0))$

4. cP reductions for *k*-colouring

To make reductions simpler we first demonstrate how to use cP rules to make the formulas *at_most_one* and *at_least_one* following the encoding we used for our solution to k-SAT. Assuming we are given $x(0), x(1), \ldots, x(i)$ and a number i we make *at_most_one* using the rule:

$$
\begin{array}{lll}
s_1 & \rightarrow_+ & s_2\ c(x(X)(-)\ x(XY1)(-)) \\
 & & \mid x(X) \\
 & & \mid x(XY1)
\end{array}
\quad (1)
$$

at_least_one requires a loop in which we create the clause:

$$
\begin{array}{llll}
s_1\ i(0) & \rightarrow_+ & s_2 & (1) \\
s_1\ c(Y) & \rightarrow_+ & s_1\ c(Y\ x(X)(+)) & (2) \\
 & & \mid x(X) & \\
 & & \mid i(X) & \\
s_1\ i(X) & \rightarrow_+ & s_2\ i(1X) & (3)
\end{array}
$$

4.1. *k*-colouring

As discussed in Section 2, the k-colouring problem is given a graph G determine if we can assign one of the k colours to each vertex such that no neighbours have the same colour. We saw that this can be solved using the formula:

$$F = \bigwedge_{v \in V} exactly\_one(x_{v,i} : i \in K) \wedge \bigwedge_{\{u,v\} \in E} \bigwedge_{i \in K} at\_most\_one(x_{u,i}, x_{v,i}) \quad (3)$$

4.1.1. *cP encoding*

To encode the problem k-colouring we shall use the following:

- Vertex v_i is encoded as $v(i)$
- Edge $e_{i,j}$ is encoded as $e(i)(j)$
- The number k is encoded as $\kappa(k)$
- The number n is encoded as $\eta(n)$
- $\sqrt{n} = x$ is encoded as $\rho(x)$

To construct the group of new variables $x(0), x(1), \ldots, x(k(n-1))$ we use the following rules:

$$s_1 \quad \rightarrow_1 \quad s_2 \mid i(X)\ k(X) \tag{1}$$

$$s_1\ j(X)\ i(Y) \quad \rightarrow_+ \quad s_1\ j(NX)\ m(X)\ i(1Y) \mid \eta(N) \tag{2}$$

$$s_2\ v(I) \quad \rightarrow_+ \quad s_3\ v(I)\ x(IX) \mid m(X) \tag{3}$$

The rules are a loop where a set of variables is created on the ith iteration, which encodes the vertices with the ith colour. To encode the ith vertex with the jth colour it is encoded as $x(n(j+i))$. The running time is $k+2$ as it loops k times on rule 2 and runs rules 1 and 3 once.

To ensure that the colours of vertices joined by an edge are not the same the following rules are used (creating a *at_most_one*):

$$\begin{aligned} s_3 \quad \rightarrow_+ \quad & s_4\ c(x(XY)(-)\ x(XY1Z)s(-)) \\ & \mid e(X)(X1Z) \\ & \mid m(Y) \end{aligned} \tag{4}$$

The rule uses the m to denote the gap between the different colours (multiples of n) and constructs the clauses such that at most, one of the variables in an edge contain that colour. The running time is 1 step.

The rules to ensure that exactly one colour is chosen for each vertex can be broken into two steps. The first is that each vertex must take at most one of the colours, which is given by:

$$\begin{aligned} s_3 \quad \rightarrow_+ \quad & s_4\ c((XY)(-)\ x(XY1Z)s(-)) \\ & \mid m(Y) \\ & \mid m(1YZ) \end{aligned} \tag{5}$$

This rule is practically the same as that given for at most one edge being the same colour. In fact the two rules can work in parallel so this has running time 1 (rules 4 and 5 combined take 1 step). The second step is that at least one colour is taken by each vertex, which is given by:

$$s_3\ v(X)\ x(X)\ i(1Y) \quad \rightarrow_+ \quad s_4\ c(x(X)s(+))\ i(Y) \tag{6}$$

$$s_4\ i(0) \quad \rightarrow_+ \quad s_5 \tag{7}$$

$$\begin{aligned} s_4\ c(Y\ x(X)(+))\ x(XZ) \quad \rightarrow_+ \quad & s_4\ c(Y\ x(X)(+)\ x(XZ)(+)) \\ & \mid m(Z) \end{aligned} \tag{8}$$

$$s_4\ i(1Y) \quad \rightarrow_+ \quad s_4\ i(Y) \tag{9}$$

These rules work in a loop over the colours. Where at the ith iteration, the ith colour is added to the clause. The looping rules, 8 and 9 run k times (they run in parallel with each other) and rules 6 and 7 once. Hence, the time taken is $k+2$.

Theorem 3. *k-colouring $\leq_p$ k-SAT in constant time.*

As demonstrated, our rules to change an instance of k-colouring to k-SAT took a time of $2k+5$. Due to k being a fixed constant and not part of the problem's input.

Corollary 4. *3-colouring is solvable in $O(\sqrt{n})$ steps.*

As 3-colouring is the instance of k-colouring for $k=3$. We know the Karp reduction from 3-colouring takes 11 steps, and solving the instance of 3-SAT takes $4\sqrt{n}+8$ steps hence the total number of steps is $O(\sqrt{n})$.

5. Conclusions

SAT is one of the most famous problems to be known to be NP-complete, with many studies using theoretical molecular computing devices to solve it [11]. As discussed in [13] many solutions to SAT have been found running in linear time using P systems.

A previous solution to SAT using cP systems also was found running in linear time [15]. However, as far as we know, our solution is the first *P system solution to run in sublinear time*. We note that as discussed in [15], many of these solutions use a variable number of rules and alphabet symbols. Our solution uses a *constant* sized alphabet and ruleset. We do, however, note that our solution uses more rules than presented in [15].

As with SAT, the 3-colouring problem has been the subject of many studies using P system variants including: cP systems [5], tissue P systems [7,20], and kernel P systems [20]. However, as far as we know, no other solution using *P systems runs in sublinear time*.

We have presented an efficient solution to the k-SAT and k-colouring problems and, as far as we know, the most efficient P system solution. Our solution to 3-colouring demonstrates that at least some of the traditional polynomial reductions can be made in a constant number of steps using cP systems. We also note that the strategies used to generate our allocations can be utilised to extend our solution to k-SAT to solving QSAT (a PSPACE complete problem).

Future work includes model-checking these solutions. We note that although cP systems have been used for model checking in the past [10] they have had the issue of memory explosion. However, if we are just model-checking a reduction this should not occur and may enable much larger instances to be model checked. Another problem is how many other problems can be efficiently reduced. The overarching problem being can we find a significantly more efficient reduction using cP systems than the traditional Turing machine reduction.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Appendix A. Distributed solution

Here we present an alternative ruleset to achieve the Cartesian product, cf. rules 9 and 10 from Table 5. This ruleset utilises *synchronous* communication between cells (see [14] for more details about multi cell communication). One of the key differences is that this ruleset always consumes something on the left-hand side. This seems to make single cell cP systems more difficult to design but also should remove the ability to produce unreasonable amounts of data in 1 step.

Our solution utilises $\sqrt{n}+1$ top-level cells with cell 0 being the main cell and all others using identical rulesets only communicating with cell 0 (cf. Fig. A.6).

Our alternative ruleset can be broken up into two parts. The first ruleset is identical for $\sqrt{n}$ cells with a processor id $p(i)$ where $i \in \{1, 2, \dots \sqrt{n}\}$. The rules:

$$\begin{array}{lll}
s_1\ ?\{X\} & \rightarrow_+ & s_2\ X \\
s_2\ a(X) & \rightarrow_+ & s_3\ a(X\ b(Y))\ \mid\ p(Y) \\
s_3\ X\ p(Y) & \rightarrow_+ & s_1\ !_0\{X\}
\end{array}
\qquad
\begin{array}{r}
(1)\\(2)\\(3)
\end{array}$$

describe the system. With each cell getting sent, a group of allocations which it then sends back with after doing a Cartesian product with its processor id.

The main cell will simply send the allocations to all of the other cells using the following rules:

$$\begin{array}{lll}
s_4\ a(X) & \rightarrow_+ & s_5\ !_{\forall}\{j(X)\} \\
s_5\ ?_{\_}\{X\} & \rightarrow_+ & s_6\ X
\end{array}
\qquad
\begin{array}{r}
(8.1)\\(8.2)
\end{array}$$

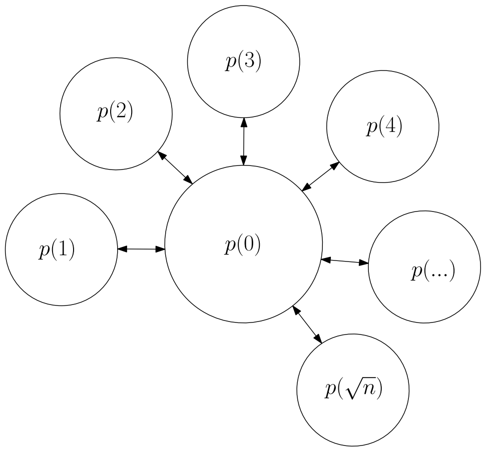


Fig. A.6. Diagram showing the graph representation of the distributed system.

Once it has received the results from the other cells it then processes them using the adjusted rules:

$$
\begin{array}{lll}
s_6 & \rightarrow_+ & s_7 \\
a(x(i(I)\ Z)\ j(Y)\ b(X)) & & \quad d(x(i(I)\ Z)\ j(\alpha(Y)\ \beta(X))) \\
 & & \quad d(x(i(IP)\ Q)\ j(\alpha(Y)\ \beta(X))) \\
 & & \mid b(x(i(IP)\ Q)\ j(X)) \\
 & & \mid m(P)
\end{array}
\tag{9'}
$$

$$
\begin{array}{lll}
s_6 & \rightarrow_+ & s_7 \\
a(x(i(I)\ Z)\ j(Y)\ b(X)) & & \quad d(x(i(I)\ Z)\ j(\alpha(Y)\ \beta(X))) \\
 & & \mid b(x(i(IP)\ Q)\ j(X)) \\
 & & \mid p(P)
\end{array}
\tag{10'}
$$

We note that the states of the original ruleset will also need to be adjusted to incorporate the changes. However, this should be straightforward.

Appendix B. Bijection between integers and branch numbers

Given an allocation of variables $\{x_0 = \alpha_0, x_1 = \alpha_1, \ldots, x_{n-1} = \alpha_{n-1}\}$ we use the following code to get branch number j:

```
$j = 0$
for $i \leftarrow 0$ to $n-1$ do
    if $\alpha_i$ = 0 then
        $j \leftarrow j * 2$
    else
        $j \leftarrow j * 2 + 1$
```

Given a branch number j we can retrieve an allocation a using the following code:

```
$a \leftarrow \{\}$
for $i \leftarrow n-1$; to 0 do
    if $j \% 2 = 0$ then
        $a \leftarrow a \cup \{x_i \leftarrow 0\}$
        $j \leftarrow j/2$
    else
        $a \leftarrow a \cup \{x_i \leftarrow 1\}$
        $j \leftarrow (j-1)/2$
```

References

[1] S. Arora, B. Barak, Computational Complexity: a Modern Approach, Cambridge University Press, 2009.
[2] C. Calude, G. Paun, Computing with Cells and Atoms: an Introduction to Quantum, DNA and Membrane Computing, CRC Press, 2000.
[3] C.S. Calude, J. Casti, M.J. Dinneen (Eds.), Unconventional Models of Computation, Springer Series in Discrete Mathematics and Theoretical Computer Science, vol. 4, Springer-Verlag, Singapore, ISBN 981-3083-69-7, 1998.
[4] S.A. Cook, The complexity of theorem-proving procedures, in: Proceedings of the Third Annual ACM Symposium on Theory of Computing, ACM, New York, NY, USA, 1971, pp. 151–158.
[5] J. Cooper, R. Nicolescu, Alternative representations of P systems solutions to the graph colouring problem, J. Membr. Comput. 1 (2019) 112–126.
[6] J. Cooper, R. Nicolescu, The Hamiltonian cycle and travelling salesman problems in cP systems, Fundam. Inform. 164 (2019) 157–180.
[7] D. Díaz-Pernil, M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez, A linear–time tissue P system based solution for the 3–coloring problem, Electron. Notes Theor. Comput. Sci. 171 (2007) 81–93.
[8] A. Henderson, R. Nicolescu, M.J. Dinneen, Solving a PSPACE-complete problem with cP systems, J. Membr. Comput. 2 (2020) 311–322, https://doi.org/10.1007/s41965-020-00064-w.
[9] M. Ionescu, G. Păun, T. Yokomori, Spiking neural P systems, Fundam. Inform. 71 (2006) 279–308.
[10] Y. Liu, R. Nicolescu, J. Sun, Formal verification of cP systems using PAT3 and ProB, J. Membr. Comput. 2 (2020) 80–94.
[11] V. Manca, DNA and membrane algorithms for SAT, Fundam. Inform. 49 (2002) 205–221.
[12] C. Martín-Vide, G. Păun, J. Pazos, A. Rodríguez-Patón, Tissue P systems, Theor. Comput. Sci. 296 (2003) 295–326.
[13] B. Nagy, On efficient algorithms for SAT, in: International Conference on Membrane Computing, in: LNCS, vol. 7762, Springer, 2012, pp. 295–310.
[14] R. Nicolescu, A. Henderson, An introduction to cP systems, in: C. Graciani, A. Riscos-Núñez, G. Păun, G. Rozenberg, A. Salomaa (Eds.), Enjoying Natural Computing: Essays Dedicated to Mario de Jesús Pérez-Jiménez on the Occasion of His 70th Birthday, Springer, 2018, pp. 204–227.
[15] R. Nicolescu, H. Wu, Complex objects for complex applications, Rom. J. Inf. Sci. Technol. 17 (2014) 46–62.
[16] G. Păun, Computing with membranes, J. Comput. Syst. Sci. 61 (2000) 108–143.
[17] G. Păun, P systems with active membranes: attacking NP-complete problems, J. Autom. Lang. Comb. 6 (2001) 75–90.
[18] M. Sipser, Introduction to the Theory of Computation, Cengage Learning, 2012.
[19] H. Stamm-Wilbrandt, Programming in Propositional Logic or Reductions: Back to the Roots (Satisfiability), Sekretariat für Forschungsberichte, Inst. für Informatik III University of Bonn, 1993.
[20] A. Turcanu, F. Ipate, Computational properties of two P systems solving the 3-colouring problem, in: 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, IEEE, 2012, pp. 62–69.