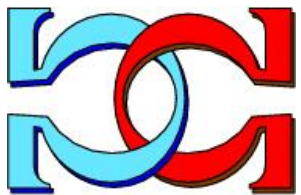
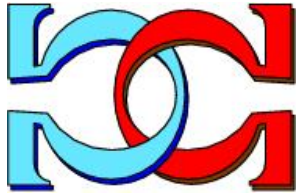
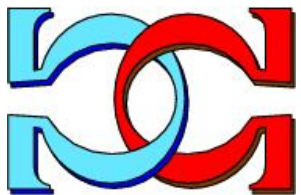


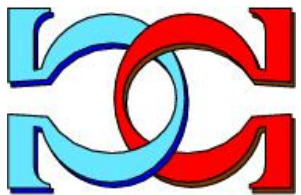
**CDMTCS
Research
Report
Series**



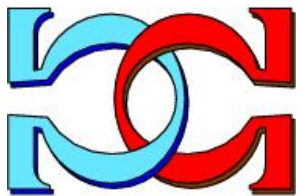
**Sublinear P system solutions
to NP-complete problems**



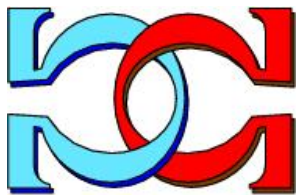
**Alec Henderson
Radu Nicolescu
Michael J. Dinneen**



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



CDMTCS-559
January 2022



Centre for Discrete Mathematics and
Theoretical Computer Science

Sublinear P system solutions to NP-complete problems

Alec Henderson, Radu Nicolescu, and Michael J. Dinneen

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

ahen386@aucklanduni.ac.nz

{r.nicolescu,m.dinneen}@auckland.ac.nz

Abstract

cP systems have been shown to very efficiently solve many NP-complete problems, i.e. 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.

Keywords: cP systems; Membrane computing; Computational complexity; NP-complete

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 [1]. P systems are an abstract model of membrane systems, with many variants being proposed such as: P systems with active membranes [2], spiking neural P systems [3], tissue P systems [4], and P systems with compound terms (cP systems) [5]. 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-complete problems efficiently such as: the Hamiltonian path, travelling salesman [6], 3-colouring [7], and subset sum [8]. 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 efficient solution to k -SAT using P and cP systems, and (2) the most efficient solution to the 3-colouring problem using a reduction of it to our k -SAT solution.

As discussed in [9], the Cook–Levin theorem gives a polynomial time reduction from any problem in NP to SAT. But for many problems, this reduction 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 the 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. The typical definition for a language to be NP-complete is that it is NP-hard and a member of NP. NP-hardness means that all languages in NP, polynomially time reduce to it. A great deal of these problems have significant practical importance. There are practicality 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 [10].

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 [11]: given a Boolean formula, does there exist a satisfying 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 variables. This restricted version is also NP-complete for any $k \geq 3$. ‘

2.1.2 Polynomial time reductions

As defined in [12], given two languages $A, B \subseteq \Sigma^*$, A is polynomial time mapping reducible (also known as Karp reducible) to a language B ($A \leq_p B$) if a polynomially computable function $f : \Sigma^* \rightarrow \Sigma^*$ 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 [11] 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 [9].

2.1.3 Useful Boolean formulas

As discussed in [9], we can make some useful formulas in CNF, which simplify the reductions to SAT. at_most_one is a formula which defines the property only one literal of the arguments is **true**:

$$at_most_one(l_1, l_2, \dots, l_n) = \bigwedge_{1 \leq i < j \leq n} (\bar{l}_i \vee \bar{l}_j)$$

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

$$at_least_one(l_1, l_2, \dots, l_n) = (l_1 \vee l_2 \vee \dots \vee l_n)$$

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

$$exactly_one(l_1, l_2, \dots, l_n) = at_most_one(l_1, l_2, \dots, l_n) \wedge at_least_one(l_1, l_2, \dots, l_n)$$

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 [9]. This more efficient implementation does, however, introduce $n - 2$ more variables. Throughout this paper, we shall assume this inefficient encoding.

2.1.4 k -colouring

As defined in [9], given a graph G with vertices V and edges E , G is k -colourable if there exists a function f :

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

Here we show the reduction from an instance of k -colouring to SAT given in [9]. The set of variables is denoted X , formula as F , and a set K as $\{1, 2, \dots, k\}$.

$$X = \{x_{v,i} : v \in V, i \in K\}$$

$$F = \bigwedge_{v \in V} exactly_one(x_{v,i} : i \in K) \wedge \bigwedge_{\{u,v\} \in E} \bigwedge_{i \in K} (\bar{x}_{u,i} \vee \bar{x}_{v,i}) \quad (1)$$

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 ($\bigwedge_{v \in V} \textit{exactly_one}(x_{v,i} : i \in K)$) 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 clauses 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 length. 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 [5] 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. *verm* 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.

Table 1: BNF grammar for cP top-cells as presented in [13].

<pre> <top-cell> ::= <state> <term> ... <state> ::= <atom> <term> ::= <atom> <sub-cell> <sub-cell> ::= <compound-term> ... <compound-term> ::= <functor> <args> ... <functor> ::= <atom> <args> ::= ‘(’ <term> ... ‘)’ </pre>

Table 2: Restricted BNF grammar for cP rules, omitting inter-cell messaging and other features not used here. α being the application mode, $\alpha \in \{0, +\}$.

```

<rule> ::= <lhs>  $\rightarrow_a$  <rhs> <promoters>
<lhs> ::= <state> <vterm> ...
<rhs> ::= <state> <vterm> ...
<promoters> ::= ('|' <vterm>) ...
<vterm> ::= <variable> | <atom> | <compound-vterm>
<compound-vterm> ::= <functor> <vargs> ...
<vargs> ::= '(' <vterm> ... ')'
```

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.

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:

$$s_1 n(X) \rightarrow_+ s_2 n(1X) \tag{1}$$

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

$$s_1 n(X) \rightarrow_1 s_2 n(1X) \tag{1}$$

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:

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)

Rule 3 is a loop that will subtract one off 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 Rule set for k -SAT

Here we assume k -SAT to be a formula in CNF with variables x_0, x_1, \dots, x_{n-1} where each clause contains at most k variables. To solve k -SAT in square root time, we break it up into three steps: generating assignment templates, generating the assignments, and finally evaluating the entire formula. Figure 11 shows a state diagram of the entire system broken down into the 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 a index of the paths from root to leaf starting at 0 for the left-most leaf (see Figure 3); 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

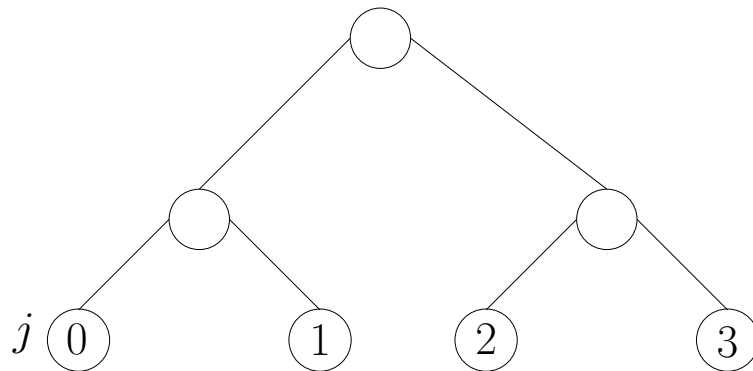


Figure 3: An example of branch numbers on a tree, where the branch number represents the path from root to leaf.

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

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 $a(0)(0)(0) a(0)(1)(1)$.

As described in [13], we can simplify the rules to evaluate a variable using lookup tables as seen in Table 4. 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 $c(x(i)(s) \dots)$ where i denotes which x_i is being referred to and s whether it is negated(−) in the clause.

For example, consider the following formula ($n = 4$, $k = 2$, and $\rho = \sqrt{n} = 2$):

$$(x_0 \vee x_1) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_1). \quad (2)$$

Table 4 contains the fixed subcells, and Table 5 the subcells that change during the evolution of the system.

Table 4: 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	$c(x(2)(−) x(1)(+))$
			$k(2)$
			$\rho(2)$

Table 5: Initial state of the subcells that *do* change whilst solving for Formula 2.

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

3.2 Allocating first \sqrt{n} variables

First, we allocate the first \sqrt{n} variables, which is then used as a lookup table. A traditional programming approach to creating these allocations can be seen in Figure 6. The outer loop is used to reference the next variable being assigned and the first inner loop creating 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.

```

1   $a \leftarrow \{(0, 0, 0), (0, 1, 1)\}$ 
2   $j \leftarrow \{0, 1, 2, 3\}$ 
3  for  $m \leftarrow 1$  to  $\sqrt{n}$  do
4       $p \leftarrow \{\}$ 
5      for  $(i, v, j)$  in  $a$  do
6           $p \leftarrow p \cup (i, v, j * 2) \cup (i, v, j * 2 + 1)$ 
7       $t \leftarrow \{\}$ 
8      for  $z$  in  $j$  do
9          if  $z \% 2 = 0$ 
10              $p \leftarrow p \cup (m, 0, z)$ 
11         else
12              $p \leftarrow p \cup (m, 1, z)$ 
13          $t \leftarrow t \cup 2 * z \cup 2 * z + 1$ 
14      $a \leftarrow p$ 
15      $j \leftarrow t$ 

```

Figure 6: Sequential algorithm for creating first \sqrt{n} allocations.

Our cP system closely models that of the sequential algorithm presented in Figure 6 with the ruleset presented in Table 7. 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 there branch numbers. Rules 3 and 4 add the next variable to the allocations. The outer loop formed by rules 1 and 6 run \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 $O(\sqrt{n})$.

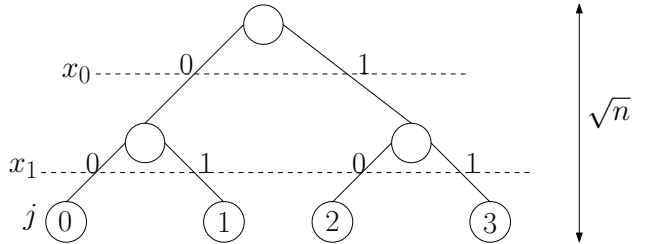
As an example, the subcells in Table 5 will change to the subcells in Figure 8 after the execution of rules 2-6.

Table 7: cP rules to allocate first \sqrt{n} variables.

$s_1 m(I)$	\rightarrow_1	$s_3 m(I)$ $ 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_+	$s_2 j(ZZ) a(Y)(0)(ZZ)$ $ m(Y)$	(3)
$s_1 j(ZZ1)$	\rightarrow_+	$s_2 j(ZZ1) a(Y)(1)(ZZ1)$ $ 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)

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	$a(1)(1)(3)$
			$m(2)$
			$j(0) j(1) j(2) j(3)$
			$j(4) j(5) j(6) j(7)$

(a) Table representation



(b) Graph representation

Figure 8: First step of the algorithm for solving Formula 2.

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 Figure 10.

A sequential version of this algorithm can be seen in Table 9. 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 $\alpha(\alpha(0)(1))(2)$.

The rules to allocate the remaining variables are in Table 12. The state diagram giving the state transitions of allocating the variables is shown in Figure 11.

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 (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 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 12 $3\sqrt{n} + 2$ steps.

For example, rule 7 will create the b subcells displayed in Table 12. 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 13 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 9: Sequential algorithm for creating the rest of the allocations.

```

1  $a \leftarrow \{(0, 0, 0), \dots\}$ 
2  $b \leftarrow \text{map } a \text{ by } (i, v, j) \rightarrow (i + \sqrt{n}, v, j)$ 
3  $p \leftarrow \{\}$ 
4 for  $m \leftarrow \sqrt{n}$  to  $n$  step  $\sqrt{n}$  do
5    $d \leftarrow \{\}$ 
6   for  $(i, v, j)$  in  $a$  do
7     for  $(y, q, x)$  in  $b$  do
8       if  $i + m = y$  then
9          $d \leftarrow d \cup (i, v, (j, x))$ 
10         $d \leftarrow d \cup (y, v, (j, x))$ 
11      for  $z$  in  $p$  do
12        if  $i + z = y$  then
13           $d \rightarrow d \cup (i, v, (j, x))$ 
14   $p \leftarrow p \cup m$ 
15   $a \leftarrow d$ 
16   $b \leftarrow \text{map } b \text{ by } (i, v, j) \rightarrow (i + \sqrt{n}, v, j)$ 

```

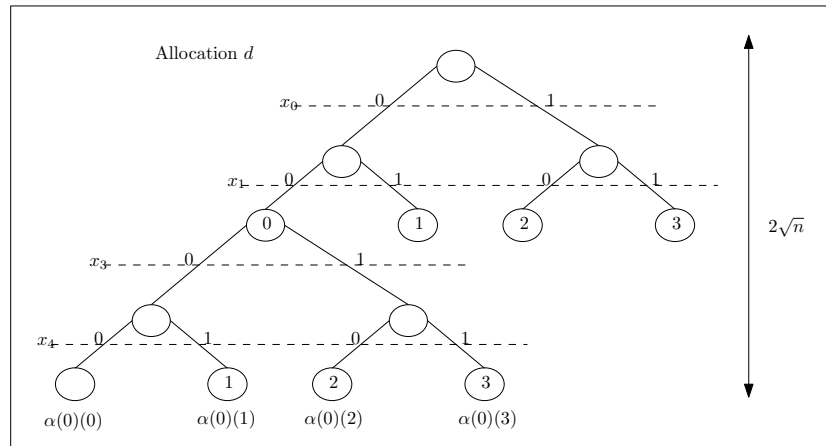
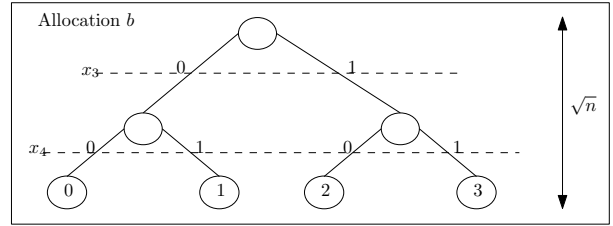
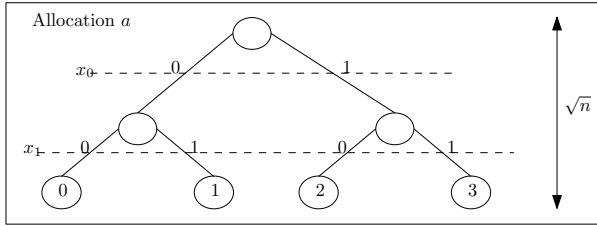


Figure 10: Diagrams representing the Cartesian product showing for $\alpha = 0$ and $n = 4$.

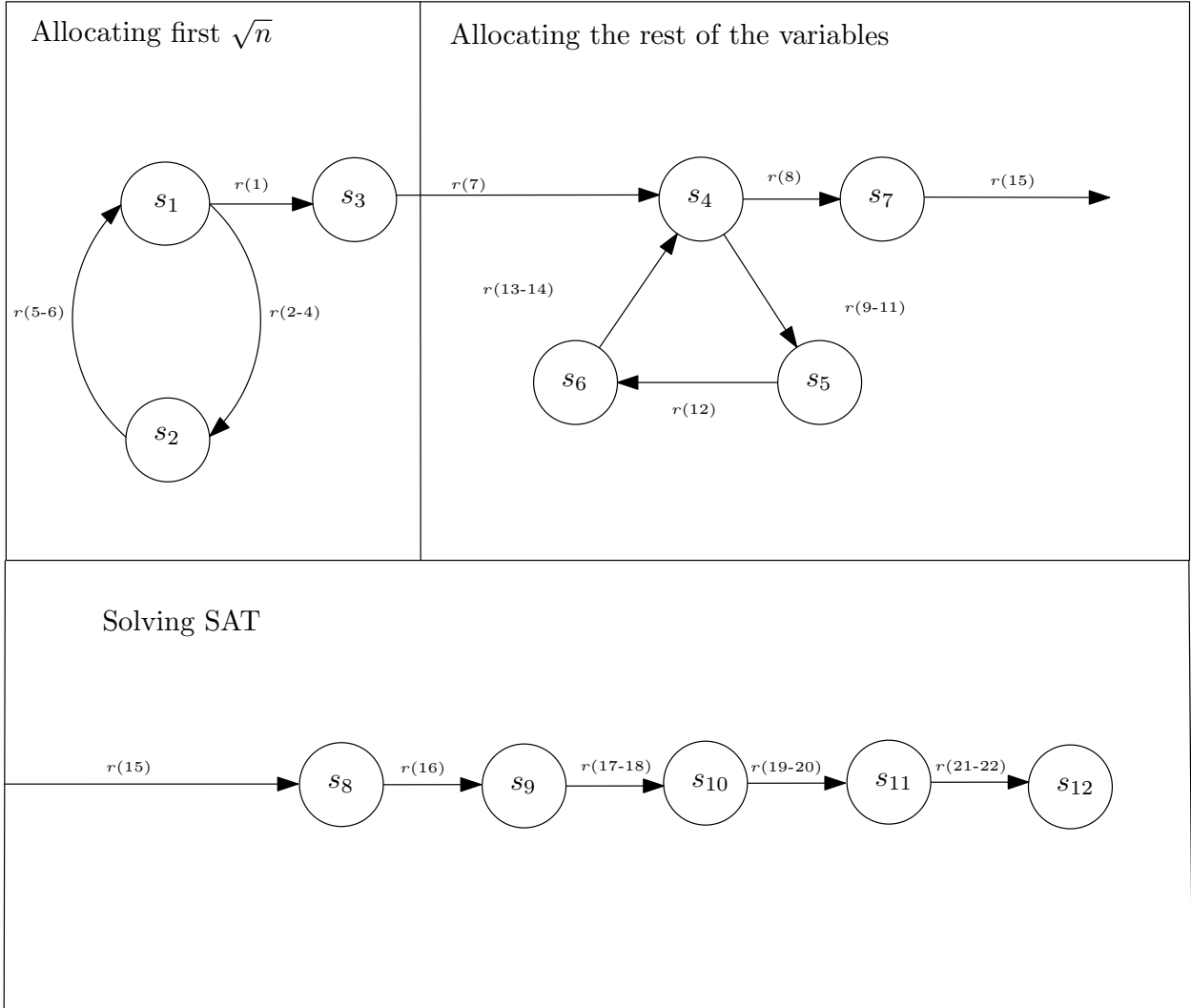


Figure 11: State diagram broken up into the three parts of the algorithm.

Table 12: Rules to create allocations.

$$s_3 a(X)(Y)(Z) \rightarrow_+ s_4 a(X)(Y)(Z) \quad (7)$$

$$\quad \quad \quad | b(XQ)(Y)(Z)$$

$$\quad \quad \quad | l(Q)$$

$$\quad \quad \quad | n(XQI)$$

$$s_4 m(QX) \rightarrow_1 s_7 m(1) \quad (8)$$

$$\quad \quad \quad | n(Q)$$

$$s_4 \rightarrow_+ s_5 d(X)(Y)(\alpha(Z)(W)) \quad (9)$$

$$\quad \quad \quad | d(XP)(V)(\alpha(Z)(W))$$

$$\quad \quad \quad | a(X)(Y)(Z)$$

$$\quad \quad \quad | b(XP)(V)(W)$$

$$\quad \quad \quad | m(P)$$

$$s_4 \rightarrow_+ s_5 d(X)(Y)(\alpha(Z)(W)) \quad (10)$$

$$\quad \quad \quad | a(X)(Y)(Z)$$

$$\quad \quad \quad | b(XP)(V)(W)$$

$$\quad \quad \quad | p(P)$$

$$s_4 m(I) \rightarrow_1 s_5 p(I) m(IQ) \quad (11)$$

$$\quad \quad \quad | l(Q)$$

$$s_5 a(X) \rightarrow_+ s_6 \quad (12)$$

$$s_6 b(X)(Y)(Z) \rightarrow_+ s_4 b(XQ)(Y)(Z) \quad (13)$$

$$\quad \quad \quad | L(Q)$$

$$\quad \quad \quad | n(XQI)$$

$$s_6 d(X) \rightarrow_+ s_4 a(X) \quad (14)$$

Table 13: Creating the next \sqrt{n} variables for Formula 2.

<i>a</i>				<i>b</i>			
<i>i</i>	<i>x_i</i>	<i>j</i>	cP representation	<i>i</i>	<i>x_i</i>	<i>j</i>	cP representation
0	0	0	$a(0)(0)(0)$	2	0	0	$b(2)(0)(0)$
1	0	0	$a(1)(0)(0)$	3	0	0	$b(3)(0)(0)$
0	0	1	$a(0)(0)(1)$	2	0	1	$b(2)(0)(1)$
1	1	1	$a(1)(1)(1)$	3	1	1	$b(3)(1)(1)$
0	1	2	$a(0)(1)(2)$	2	1	2	$b(2)(1)(2)$
1	0	2	$a(1)(0)(2)$	3	0	2	$b(3)(0)(2)$
0	1	3	$a(0)(1)(3)$	2	1	3	$b(2)(1)(3)$
1	1	3	$a(1)(1)(3)$	3	1	3	$b(3)(1)(3)$

<i>d</i>			
<i>i</i>	<i>x_i</i>	<i>j</i>	cP representaiion
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	$\alpha(0)(0)$	$d(3)(0)(\alpha(0)(0))$

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 and checked whether an allocation exists that satisfies the formula. A sequential algorithm describing the steps taken by our cP system can be found in Figure 14. A state diagram of this final part of the algorithm is shown in Figure 11.

The rule set presented in Table 15 is the cP system equivalent of the algorithm presented in Figure 14. 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 exist, 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 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). Making the running time of the rule set presented in Table 15 5 steps.

For example, if we have the allocation displayed in Table 16 rule 16 will create a clause for each of the allocations resulting in subcells denoted κ with a branch number as seen in Table 17. Once created, these clauses are evaluated, using rules 17 and 18 as shown in Table 18. After the evaluation, the clauses with matching j are combined using an **and** operation \wedge (rules 22 and 23) as shown in Table 19. Finally, the system checks if there is any f subcell containing a one; if there is, then there exists a satisfying assignment.

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

The rule sets presented in Tables 7, 12 and 15 solve k -SAT with the running times being $\sqrt{n} + 1$, $3\sqrt{n} + 2$ and 5 making the total time $4\sqrt{n} + 8$, $O(\sqrt{n})$.


```

1   $a \leftarrow \{(0, 0, (\alpha(\dots)(0))), \dots\}$ 
2   $c \leftarrow \{(i, s), (j, t), \dots, \dots\}$ 
3  // a set of  $k$  tuples with each  $k$  tuple item being a pair.
4   $\gamma \leftarrow \{\}$ 
5  for  $(i, v, j)$  in  $a$  do
6       $\gamma \leftarrow \gamma \cup j$ 
7   $\kappa \leftarrow \{\}$ 
8  for  $d$  in  $c$  do
9      for  $j$  in  $\gamma$  do
10          $\kappa \leftarrow \kappa \cup (d, j)$ 
11   $t \leftarrow \{\}$ 
12  for  $(d, j)$  in  $\kappa$  do
13      $p \leftarrow 0$ 
14     for  $(i, s)$  in  $d$  do
15         for  $(x, v, y)$  in  $a$  do
16             if  $x = i$  and  $j = y$  and  $y(v, s) = 1$  then
17                  $p \leftarrow 1$ 
18          $t \leftarrow t \cup (p, j)$ 
19   $\kappa \leftarrow t$ 
20   $f = \{\}$ 
21  for  $y$  in  $\gamma$  do
22      $v \leftarrow 1$ 
23     for  $(p, j)$  in  $\kappa$  do
24         if  $p = 0$  and  $j = y$  then
25              $v \leftarrow 0$ 
26      $f \leftarrow f \cup v$ 
27   $r \leftarrow 0$ 
28  for  $v$  in  $f$  do
29     if  $v = 1$  then
30          $r \leftarrow 1$ 

```

Figure 14: 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 15: Rules to solve using the allocations.

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

Table 16: 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 17: 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, +	$\alpha(0) \beta(0)$	$\kappa(x(0)(-) x(1)(+) j(\alpha(0)(0)))$

Table 18: 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 19: 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 $x(0), x(1), \dots, x(i)$ and a number i we make *at_most_one* using the rule :

$$\begin{array}{l}
 s_1 \rightarrow_+ s_2 \ c(x(X)(-) \ x(XY1)(-)) \\
 \quad \quad \quad | \ x(X) \\
 \quad \quad \quad | \ x(XY1)
 \end{array} \tag{1}$$

at_least_one requires a loop in which we create the clause:

$$s_1 \ i(0) \rightarrow_+ s_2 \tag{1}$$

$$\begin{array}{l}
 s_1 \ c(Y) \rightarrow_+ s_1 \ c(Y \ x(X)(+)) \\
 \quad \quad \quad | \ x(X)
 \end{array} \tag{2}$$

$$\begin{array}{l}
 \quad \quad \quad | \ i(X) \\
 s_1 \ i(X) \rightarrow_+ s_2 \ i(1X)
 \end{array} \tag{3}$$

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:

$$F = \bigwedge_{v \in V} \textit{exactly_one}(x_{v,i} : i \in K) \wedge \bigwedge_{\{u,v\} \in E} \bigwedge_{i \in K} (\bar{x}_{u,i} \vee \bar{x}_{v,i}) \tag{3}$$

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 $x(0), x(1), \dots, x(k(n-1))$ we use the following rules:

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

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

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

The rules are a loop where a set of variables is created on the i th iteration, which encodes the vertices with the i th colour. To encode the i th vertex with the j th 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*):

$$s_3 \quad \rightarrow_+ \quad s_4 \ c(x(XY)(-) \ x(XY1Z)s(-)) \quad (4)$$

$$\mid e(X)(X1Z)$$

$$\mid m(Y)$$

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 being that each vertex must take at most one of the colours which is given by:

$$s_3 \quad \rightarrow_+ \quad s_4 \ c((XY)(-) \ x(XY1Z)s(-)) \quad (5)$$

$$\mid m(Y)$$

$$\mid m(1YZ)$$

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 (when totalling rules 4 and 5 is total 1 step). The second step being that at least one colour is taken by each vertex which is given by:

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

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

$$s_4 c(Y x(X)(+)) x(XZ) \quad \rightarrow_+ \quad s_4 c(Y x(X)(+) x(XZ)(+)) \quad (8)$$

$$\quad \quad \quad | \quad m(Z)$$

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

These rules work in a loop over the colours. Whereat the i th iteration, the i th 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 Discussion

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 [14]. As discussed in [15] many solutions to SAT have been found running in linear time using P systems.

A previous solution using cP systems also was found running in linear time [16]. 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 [16], 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 [16].

As with SAT, the 3-colouring problem has been the subject of many studies using P system variants including: cP systems [7], tissue P systems [17, 18], and kernel P systems [18]. However, as far as we know, no other solution using *P systems runs in sublinear time*.

6 Conclusions

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 [8] 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.

References

- [1] G. Păun, Computing with membranes, *Journal of Computer and System Sciences* 61 (1) (2000) 108–143.
- [2] G. Păun, P systems with active membranes: Attacking NP-Complete problems, *Journal of Automata, Languages and Combinatorics* 6 (1) (2001) 75–90.
- [3] M. Ionescu, G. Păun, T. Yokomori, Spiking neural P systems, *Fundamenta informaticae* 71 (2, 3) (2006) 279–308.
- [4] C. Martín-Vide, G. Păun, J. Pazos, A. Rodríguez-Patón, Tissue P systems, *Theoretical Computer Science* 296 (2) (2003) 295–326.
- [5] 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*, LNCS 11270, Springer, 2018, pp. 204–227.
- [6] J. Cooper, R. Nicolescu, The Hamiltonian cycle and travelling salesman problems in cP systems, *Fundamenta Informaticae* 164 (2-3) (2019) 157–180.
- [7] J. Cooper, R. Nicolescu, Alternative representations of P systems solutions to the graph colouring problem, *Journal of Membrane Computing* 1 (2) (2019) 112–126.
- [8] Y. Liu, R. Nicolescu, J. Sun, Formal verification of cP systems using PAT3 and ProB, *Journal of Membrane Computing* 2 (2) (2020) 80–94.
- [9] 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.
- [10] S. Arora, B. Barak, *Computational complexity: a modern approach*, Cambridge University Press, 2009.
- [11] S. A. Cook, The complexity of theorem-proving procedures, in: *Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71*, ACM, New York, NY, USA, 1971, p. 151–158. [doi:10.1145/800157.805047](https://doi.org/10.1145/800157.805047).
- [12] M. Sipser, *Introduction to the Theory of Computation*, Cengage Learning, 2012.

- [13] A. Henderson, R. Nicolescu, M. J. Dinneen, Solving a PSPACE-complete problem with cP systems, *Journal of Membrane Computing* 2 (4) (2020) 311–322. doi:10.1007/s41965-020-00064-w.
- [14] V. Manca, DNA and membrane algorithms for SAT, *Fundamenta Informaticae* 49 (1-3) (2002) 205–221.
- [15] B. Nagy, On efficient algorithms for SAT, in: *International Conference on Membrane Computing*, LNCS 7762, Springer, 2012, pp. 295–310.
- [16] R. Nicolescu, H. Wu, Complex objects for complex applications, *Romanian Journal of Information Science and Technology* 17 (1) (2014) 46–62.
- [17] 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, *Electronic Notes in Theoretical Computer Science* 171 (2) (2007) 81–93.
- [18] A. Turcanu, F. Ipatu, 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.

A Distributed solution

Here we present an alternative ruleset to achieve the Cartesian product, cf. rules 9 and 10 from Table 12. This ruleset utilises *synchronous* communication between cells (see [5] 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 rule sets only communicating with cell 0 (cf Figure 20).

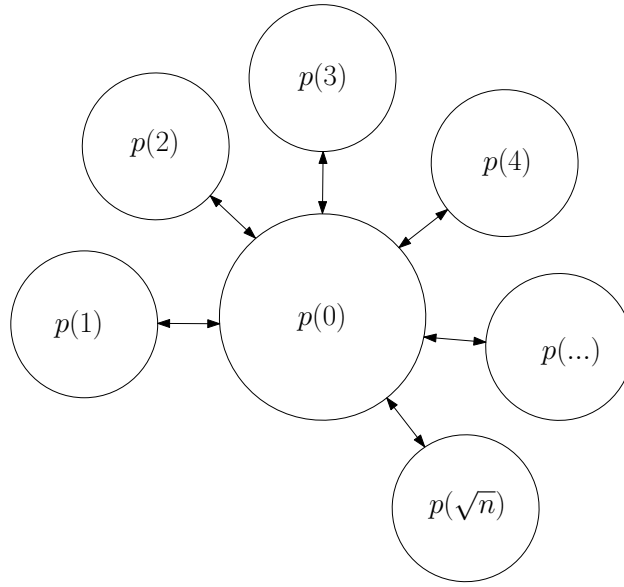


Figure 20: Diagram showing the graph representation of the distributed system.

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:

$$s_1 ?\{X\} \rightarrow_+ s_2 X \quad (1)$$

$$s_2 a(X) \rightarrow_+ s_3 a(X b(Y)) \mid p(Y) \quad (2)$$

$$s_3 X p(Y) \rightarrow_+ s_1 !_0\{X\} \quad (3)$$

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:

$$s_4 a(X) \rightarrow_+ s_5 !_\forall\{j(X)\} \quad (8.1)$$

$$s_5 ?_{-}\{X\} \rightarrow_+ s_6 X \quad (8.2)$$

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

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

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

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

B Bijection between integers and branch numbers

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

```

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

```

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

```

1  $a \leftarrow \{\}$ 
2 for  $i \leftarrow n - 1$ ; to  $0$  do
3   if  $j \% 2 = 0$  then
4      $a \leftarrow a \cup \{x_i \leftarrow 0\}$ 
5      $j \leftarrow j / 2$ 
6   else
7      $a \leftarrow a \cup \{x_i \leftarrow 1\}$ 
8      $j \leftarrow (j - 1) / 2$ 

```