Fundamenta Informaticae XXI (2001) 1001–1023
DOI 10.3233/FI-2015-0000
IOS Press

The Hamiltonian Cycle and Travelling Salesman Problems in cP systems

James Cooper [C]
Department of Computer Science
University of Auckland
Auckland, New Zealand
jcoo092@aucklanduni.ac.nz

Radu Nicolescu
Department of Computer Science
University of Auckland
Auckland, New Zealand

Abstract. The Hamiltonian Cycle Problem (HCP) and Travelling Salesman Problem (TSP) are long-standing and well-known NP-hard problems. The HCP is concerned with finding paths through a given graph such that those paths visit each node exactly once after the start, and end where they began (i.e., Hamiltonian cycles). The TSP builds on the HCP and is concerned with computing the lowest cost Hamiltonian cycle on a weighted (di)graph. Many solutions to these problems exist, including some from the perspective of P systems. For the TSP however, almost all these papers have combined membrane computing with other approaches for approximate solution algorithms, which is surprising given the plethora of P systems solutions to the HCP. A recent paper presented a brute-force style P systems solution to the TSP with a time complexity of $\mathcal{O}(n^2)$, exploiting the ability of P systems to reduce time complexity in exchange for space complexity, but the resultant system had a fairly high number of rules, around 50. Inspired by this paper, and seeking a more concise representation of an exact brute-force TSP algorithm, we have devised a P systems algorithm based on cP systems (P systems with Complex Objects) which requires five rules and takes $n + 3$ steps. We first provide some background on cP systems and demonstrate a fast new cP systems method to

Address for correspondence: Department of Computer Science, The University of Auckland, Private Bag 92019, Auckland 1142, New Zealand
[C]Corresponding author

find the minimum of a multiset, then describe our solution to the HCP, and build on that for our TSP algorithm. This paper describes said algorithms, and provides an example application of our TSP algorithm to a given graph and a digraph variant.

Keywords: cP systems, P systems, Prolog terms and unification, Travelling Salesman Problem, Hamiltonian Cycle Problem

1. Introduction

The Hamiltonian Path Problem (HPP) is a long-standing, well-known computationally hard (NP-hard) problem in Computer Science and related fields. The problem consists of trying to determine whether for a given graph (digraph) there exists a traversal of the graph in which each node (vertex) is visited exactly once. Depending on the particular problem at hand, the starting and/or ending node may or may not be arbitrarily selected. Intrinsically linked, and arguably more useful, is *describing* such a Hamiltonian path – describing such a path proves the existence of it, and allows for its potential use in practical applications. The Hamiltonian Cycle Problem (HCP) is a closely related variant, in which the Hamiltonian path must also be a cycle, i.e., the final node (vertex) in the path is the same as the starting one, such that every node (vertex) is visited exactly once after the start.

The HPP/HCP has been examined before through the lens of P systems, with different variants of P systems used such as Tissue P systems [1], P systems with active membranes [2, 3], Recogniser P systems [4], Asynchronous P systems [5] and Spiking Neural P systems [6]. Jiménez *et al.* used the HPP as an example computation when discussing complexity classes in P systems [7]. In fact, it was demonstrated relatively early in the history of P systems that some variants at least could be used to solve the HPP in linear time [8]. This last approach, using P systems with membrane creation, is arguably the closest to our approach, which involves the instantiation of subcells (complex objects).

The Travelling Salesman Problem (TSP) is an extension upon the HCP, and is about finding the minimum cost Hamiltonian cycle in a *weighted* graph. It has been described as analogous to finding the shortest route for a travelling salesman to visit multiple cities in one trip (whence the name). The problem is arguably the most useful of the three to solve in practice, and has been studied extensively, spawning many papers, dissertations and books on the topic (e.g. [9–12] amongst many, many others). A variety of sophisticated algorithms have been developed to solve the problem efficiently, in either the exact or approximate case. This paper does not seek overturn this prior body of work. Instead, it seeks to address the problem from a cP systems perspective. Note that we discuss here the *optimisation* form of the TSP, which is NP-Hard but not NP-Complete, and not the *decision* form of the TSP, which is NP-Complete.[1] This demonstrates that cP systems, and presumably other forms of P systems too, are capable of solving in polynomial time problems for which there is no known traditional polynomial-time verification

A small amount of work has been done on the TSP from the perspective of membrane computing, beginning with the work of Nishida [13], who used a combination of a membrane structure and pre-existing methods to search for an approximate solution in a space of solutions to the TSP for a given digraph. Others built on this approach by integrating techniques such as Genetic Algorithms [14, 15], Ant Colony Optimisation [16] and Active Evolution [17], along with more complex approaches for

[1] See for example `https://www.ibm.com/developerworks/community/blogs/jfp/entry/no_the_tsp_isn_t_np_complete?lang=en` for an interesting but informal discussion of the difference.

multiple salesman problems [18]. A paper by Chen *et al.* [19] was apparently also written on the topic, but no copy of that paper could be located.

Surprisingly however, given the multitude of solutions already demonstrated for the HCP, all these TSP papers have been written from the perspective of approximate solutions to the TSP. They typically take an approach of using membranes to divide up the search space of potential solutions, whilst applying other pre-existing techniques to the process. These papers have used membranes to structure a search space, but in our view have not fully embraced the P systems model, e.g. none of them have specified typical P systems rewriting rules, instead applying other techniques within the subcells, and using P systems rules only to move potential solutions between cells. More recently however, Guo and Dai presented a paper on solving the TSP in the exact case using cell-like P systems [20], requiring roughly 50 rules. By exploiting the well-known property of many P systems variants that time complexity can often be exchanged for space and/or processing complexity [7, 21–23], the authors derived a P systems algorithm that can solve the TSP in $\mathcal{O}(n^2)$ time.

Inspired by [20], we derive a cP systems algorithm for solving the TSP, fully exploiting the power of P systems' theoretical infinite resources, as well as the compactness of representation of cP systems. Such systems have been described extensively in prior papers, in particular most recently in [24]. The use of cP systems' generic maximally parallel multiset rewriting rules, and Prolog-like terms and unification, increases the expressive power of each rule, enabling us to specify here a fixed-size ruleset of five rules, applicable to digraphs of any size or complexity. Further, our algorithm can solve any instance in $n + 3$ steps, where n is the number of vertices in the digraph (thus the time complexity of our algorithm is $\mathcal{O}(n)$). This paper mainly focuses on the TSP, though we begin with a solution to the less-complicated HPP, then expand that to the HCP, and then expand upon that to solve the TSP. In the latter two cases, only relatively minor modifications to our rules are required at each step in order to take account of the stricter requirements of the problem.

For the sake of space, we hereafter assume that the reader is familiar with the basic concepts of P systems (see [25] for a good, if slightly old, introduction to P systems), and the HCP & the TSP. In Section 2 we present revised and improved material from previous papers that describes key aspects of cP systems relevant to this paper (the reader is referred to [24, 26, 27] also), before setting out a new cP systems method for finding the minimum of a multiset in a single step, which requires two rules. Our algorithms, and in particular the rules for them, are presented for the HPP & HCP and TSP cases respectively in Sections 3 and 4. We provide worked examples for the TSP in Section 5, applying our algorithm to a specific weighted graph and a modified digraph version of it. We then consider some potential variations on the algorithm for differing results in Section 6. We conclude the paper and suggest some possible directions for future work in Section 7. Finally, we briefly discuss simulations of the TSP system written in SWI-Prolog, F# and Erlang in Appendix A.

2. cP Systems : P Systems with Complex Symbols

In the interests of self-containment, we present here some material describing the background of cP systems, for the benefit of readers as yet unfamiliar with the topic. More extensive presentation of cP systems has appeared most recently in [24], and it is recommended that the interested reader peruse that paper as well. There are two notable additions shown here that are not in [24], however: the stronger semantics for inhibitors, to fully implement logical negation; and the minimum-finding algorithm explained

in Section 2.4, used in solving the TSP. We wish to point out that, while cP systems is transitively bio-inspired through its basis in P systems, it has not been developed with the aim of simulating or modelling real-world biology, and instead is intended as a useful theoretical model for computation.

2.1. Complex symbols as subcells

Complex symbols or *subcells*, play the roles of cellular micro-compartments or substructures, such as organelles, vesicles or cytoophidium assemblies ("snakes"), which are embedded in cells or travel between cells, but without having the full processing power of a complete cell. In our proposal, *subcells* represent nested labelled data compartments with no processing power of their own; instead, they are acted upon by the rules of their enclosing cells.

Our basic vocabulary consists of *atoms* and *variables*, collectively known as *simple symbols*. *Complex symbols* are similar to Prolog-like *first-order terms*, recursively built from *multisets* of atoms and variables. Together, complex symbols and simple symbols (atoms, variables) are called *symbols*, and can be defined by the following formal grammar:

```
<symbol> ::= <atom> | <variable> | <term>
<term> ::= <functor> '(' <argument> ')'
<functor> ::= <atom>
<argument> ::= λ | ( <symbol> )+
```

Atoms are typically denoted by lower case letters (or, occasionally, digits), such as a, b, c, *1*. *Variables* are typically denoted by uppercase letters, such as X, Y, Z. *Functors* are term (subcell) labels; here functors can only be atoms, not variables.

For improved readability, we also consider *anonymous variables*, which are denoted by underscores ("_"). Each underscore occurrence represents a *new* unnamed variable and indicates that something, in which we are not interested, must fill that slot.

Symbols that do *not* contain variables are called *ground*, e.g.:

- Ground symbols: a, $a(\lambda)$, $a(b)$, $a(bc)$, $a(b^2c)$, $a(b(c))$, $a(bc(\lambda))$, $a(b(c)d(e))$, $a(b(c)d(e))$, $a(b(c)d(e(\lambda)))$, $a(bc^2d)$.
- Symbols which are not ground: X, $a(X)$, $a(bX)$, $a(b(X))$, $a(XY)$, $a(X^2)$, $a(XdY)$, $a(Xc())$, $a(b(X)d(e))$, $a(b(c)d(Y))$, $a(b(X^2)d(e(Xf^2)))$; also, using anonymous variables: $\_$, $a(b\_)$, $a(X\_)$, $a(b(X)d(e(\_)))$.
- This term-like construct which starts with a variable is not a symbol (this grammar defines first-order terms only): $X(aY)$.

Note that we may abbreviate the expression of complex symbols by removing inner λ's as explicit references to the empty multiset, e.g. $a(\lambda) = a()$.

In *concrete* models, *cells* may contain *ground* symbols only (no variables). Rules may however contain *any* kind of symbols, atoms, variables and terms (whether ground or not).

Unification. All symbols which appear in rules (ground or not) can be (asymmetrically) *matched* against *ground* terms, using an ad-hoc version of *pattern matching*, more precisely, a *one-way first-order syntactic unification* (one-way, because cells may not contain variables). An atom can only match another

copy of itself, but a variable can match any multiset of ground terms (including λ). This may create a combinatorial *non-determinism*, when a combination of two or more variables are matched against the same multiset, in which case an arbitrary matching is chosen. For example:

- Matching $a(b(X)fY) = a(b(cd(e))f^2g)$ deterministically creates a single set of unifiers: $X, Y = cd(e), fg$.
- Matching $a(XY^2) = a(de^2f)$ deterministically creates a single set of unifiers: $X, Y = df, e$.
- Matching $a(b(X)c(1X)) = a(b(1^2)c(1^3))$ deterministically creates one single unifier: $X = 1^2$.
- Matching $a(b(X)c(1X)) = a(b(1^2)c(1^2))$ fails.
- Matching $a(XY) = a(df)$ non-deterministically creates one of the following four sets of unifiers: $X, Y = \lambda, df$; $X, Y = df, \lambda$; $X, Y = d, f$; $X, Y = f, d$.

2.2. High-level or generic rules

Typically, our rules use *states* and are applied top-down, in the so-called *weak priority* order.

Pattern matching. Rules are matched against cell contents using the aforementioned *pattern matching*, which involves the rule's left-hand side, promoters and inhibitors – promoters and inhibitors are further discussed below, in a following paragraph.

Generally, variables have *global rule scope*; these are assumed to be introduced by *existential* quantifiers preceding the rule – with the exception of inhibitors, which may introduce *local variables*, as further discussed below.

The matching is *valid* only if, after substituting variables by their values, the rule's right-hand side contains ground terms only (so *no* free variables are injected in the cell or sent to its neighbours), as illustrated by the following sample scenario:

- The cell's *current content* includes the *ground term*:
  $n(a\,\phi(b\,\phi(c)\,\psi(d))\,\psi(e))$
- The following (state-less) *rewriting rule* is considered:
  $n(X\,\phi(Y\,\phi(Y_1)\,\psi(Y_2))\,\psi(Z)) \;\rightarrow\; v(X)\;n(Y\,\phi(Y_2)\,\psi(Y_1))\;v(Z)$
- Our pattern matching determines the following *unifiers*:
  $X = a, Y = b, Y_1 = c, Y_2 = d, Z = e$.
- This is a *valid* matching and, after *substitutions*, the rule's *right-hand* side gives the *new content*:
  $v(a)\;n(b\,\phi(d)\,\psi(c))\;v(e)$

Generic rules format. We consider rules of the following *generic* format (we call this format generic, because it actually defines templates involving variables):

$$\textit{current-state}\ \ \textit{symbols}\ldots\ \ \rightarrow_{\alpha}\ \ \textit{target-state}\ \ (\textit{in-symbols})\ldots\ \ (\textit{out-symbols})_{\delta}\ldots\ \ |\ \textit{promoters}\ldots\ \ \neg\ \textit{inhibitors}\ldots$$

Where:

- *current-state* and *target-state* are atoms or terms;
- *symbols*, *in-symbols*, *promoters* and *inhibitors* are symbols;
- *in-symbols* become available after the end of the current step only, as in traditional P systems (we can imagine that these are sent via an ad-hoc fast *loopback* channel);
- subscript $\alpha \in \{\texttt{min}, \texttt{max}\}$, indicates the application mode, as further discussed in the example below;
- *out-symbols* are sent, at the end of the step, to the cell's structural neighbours. These symbols are enclosed in round parentheses which further indicate their destinations, above abbreviated as δ. The most usual scenarios include:
  - $(a) \downarrow_i$ indicates that a is sent over outgoing arc i (unicast);
  - $(a) \downarrow_{i,j}$ indicates that a is sent over outgoing arcs i and j(multicast);
  - $(a) \downarrow_{\forall}$ indicates that a is sent over all outgoing arcs (broadcast).

  All symbols sent via one *generic rule* to the same destination form one single *message* and they travel together as one single block (even if the generic rule is applied in mode `max`).

Promoters and inhibitors. To define additional useful matchings expressively, our promoters and inhibitors may also use virtual "equality" terms, written in infix format, with the $=$ operator. For example, including the term $(ab = XY)$ indicates the following additional matching constraints on variables X and Y: either $X, Y = ab, \lambda$; or $X, Y = a, b$; or $X, Y = b, a$; or $X, Y = \lambda, ab$.

To usefully define inhibitors as logical negations, variables which only appear in the scope of an inhibitor are assumed to have *local scope*. These variables are assumed to be defined by *existential* quantifiers, immediately after the negation. Semantically, this is equivalent as introducing these variables at the global rule level, but by *universal* quantifiers, after all other global variables, which are introduced by *existential* quantifiers.

As an illustration, consider a cell containing $a(c)\ a(ccc)$ and contrast two rules, containing the following two sample promoter/inhibitor pairs (for brevity, other rule details are omitted here).

$$\ldots\ \ |\ \ a(cXY)\ \ \neg\ \ a(X) \qquad (1)$$
$$\ldots\ \ |\ \ a(cZ)\ \ \neg\ \ (Z = XY)\ \ a(X) \qquad (2)$$

These two rules appear quite similar and their inhibitor tests share the same expression: NO $a(X)$ may be present in the cell.

Rule (1) uses two global variables, X, Y. According to its promoter, $a(cXY)$, these variables can be matched in four different ways: (1a) $X, Y = \lambda, \lambda$; (1b) $X, Y = cc, \lambda$; (1c) $X, Y = \lambda, cc$; (1d) $X, Y = c, c$. Three different unifications, (1a), (1b), (1c), pass the inhibitor test, as there are no cell terms $a()$, $a(cc)$, $a()$, respectively. Unification (1d) fails the inhibitor test, because there IS one cell term $a(c)$.

Rule (2) uses one global variable, Z, and two local inhibitor variables, X, Y. According to its promoter, $a(cZ)$, variable Z can be matched in two different ways: (2a) $Z = \lambda$; (2b) $Z = cc$. Unification (2a) passes the inhibitor test, because it only generates one local unification, $X, Y = \lambda, \lambda$, and there is NO cell term $a()$. Unification (2b) fails the inhibitor test, because it generates all the following three local unifications: (2b1) $X, Y = cc, \lambda$; (2b2) $X, Y = \lambda, cc$; (2b3) $X, Y = c, c$; and there IS a cell term corresponding to (2b3), $a(c)$.

The pattern of rule (2) will be used later, in Section 2.4, to define a single step minimum-finding ruleset.

Application modes – `min` and `max`. To explain our two rule application modes, `min` and `max`, let us consider a cell, σ, containing three counter-like complex symbols, $c(1^2)$, $c(1^2)$, $c(1^3)$, and the two possible application modes of the following high-level "decrementing" rule:

$$(\rho_\alpha)\ S_1\ c(1\,X) \to_\alpha S_2\ c(X), \text{where } \alpha \in \{\texttt{min},\texttt{max}\}.$$

The left-hand side of rule ρ_α, $c(1\,X)$, can be unified in three different ways, to each one of the three c symbols extant in cell σ. Conceptually, we instantiate this rule in three different ways, each one tied and applicable to a distinct symbol:

$$\begin{aligned}
(\rho_1) \quad & S_1\ c(1^2) \to S_2\ c(1),\\
(\rho_2) \quad & S_1\ c(1^2) \to S_2\ c(1),\\
(\rho_3) \quad & S_1\ c(1^3) \to S_2\ c(1^2).
\end{aligned}$$

1. If $\alpha = \texttt{min}$, rule $\rho_{\texttt{min}}$ non-deterministically selects and applies one of these virtual rules ρ_1, ρ_2, ρ_3. Using ρ_1 or ρ_2, cell σ ends with counters $c(1)$, $c(1^2)$, $c(1^3)$. Using ρ_3, cell σ ends with counters $c(1^2)$, $c(1^2)$, $c(1^2)$.

2. If $\alpha = \texttt{max}$, rule $\rho_{\texttt{max}}$ applies in parallel all these virtual rules ρ_1, ρ_2, ρ_3. Cell σ ends with counters $c(1)$, $c(1)$, $c(1^2)$.

Semantically, the `max` mode is equivalent to a virtual sequential while loop around the same rule in `min` mode, which is repeated until it is no more applicable. Note, however, that all such applications of the rule are carried out concurrently in a single step.

Special cases. Simple scenarios involving generic rules are sometimes semantically equivalent to sets of non-generic rules defined via bounded loops. For example, consider the rule

$$S_1\ a(x(I)\ y(J)) \ \to_{\texttt{max}}\ S_2\ b(I)\ c(J),$$

where the cell's contents guarantee that I and J only match integers in ranges $[1, n]$ and $[1, m]$, respectively. Under these assumptions, this rule is equivalent to the following set of non-generic rules:

$$S_1\ a_{i,j} \ \to S_2\ b_i\ c_j,\ \forall i \in [1, n], j \in [1, m].$$

However, unification is a much more powerful concept, which cannot be generally reduced to simple bounded loops.

Benefits. This type of generic rules allows (i) a reasonably fast parsing and processing of subcomponents, and (ii) algorithm descriptions with *fixed-size alphabets* and *fixed-sized rulesets*, independent of the size of the problem and number of cells in the system (often *impossible* with only atomic symbols).

2.3. Data structures in cP systems

In this section we sketch the design of two high-level data structures, similar to the data structures used in high-level pseudocode or programming languages: natural numbers and lists, together with alternative more legible notations

Natural numbers. Natural numbers can be represented via *multisets* containing repeated occurrences of the *same* atom. For example, considering that *1* represents an ad-hoc unary digit, the following complex symbols can be used to describe the contents of a virtual integer *variable* a: $a() = a(\lambda)$ — the value of a is 0; $a(1^3)$ — the value of a is 3. For concise expressions, we may alias these number representations by their corresponding numbers, e.g. $a() \equiv a(0), b(1^3) \equiv b(3)$. Nicolescu et al. [26, 27] show how the basic arithmetic operations can be efficiently modelled by P systems with complex symbols.

Here follows a list of simple arithmetic expressions, assignments and comparisons:

$$
\begin{array}{lr}
x = 0 \;\equiv\; x(\lambda) & \\
x = 1 \;\equiv\; x(1) & \\
x = 2 \;\equiv\; x(11) & \\
x = n \;\equiv\; x(1^n) & \\
 & \\
x \leftarrow y + z \;\equiv\; y(Y)\, z(Z) \;\rightarrow\; x(YZ) & \textit{destructive add} \\
x \leftarrow y + z \;\equiv\; \rightarrow\; x(YZ) \;\mid\; y(Y)\, z(Z) & \textit{preserving add} \\
 & \\
x = y \;\equiv\; x(X)\, y(X) & \textit{equality} \\
x \leq y \;\equiv\; x(X)\, y(XY) & \textit{less than or equal to} \\
x < y \;\equiv\; x(X)\, y(X1Y) & \textit{strictly less than}
\end{array}
$$

Note that strictly less than ($<$) requires the extra 1, because Y can match on λ.

Lists. Consider the *list* y, containing the following sequence of values: $[u; v; w]$. List y can be represented as the complex symbol $y(\,\gamma(u\ \gamma(v\ \gamma(w\ \gamma()))))$, where the ad-hoc atom γ represents the list constructor *cons* and $\gamma()$ the empty list. We may also alias this list by the more expressive equivalent notation $y(u \,|\, v \,|\, w)$ – or by $y(u \,|\, y')$, $y'(v \,|\, w)$ – where operator $|$ separates the head and the tail of the list. The notation $z(|)$ is shorthand for $z(\gamma())$ and indicates an empty list, z.

2.4. Efficient minimum-finding with cP rules

Consider an unstructured multiset $A \subseteq \mathbb{N}$ of size n. It is well known that (1) any sequential algorithm that finds its minimum needs at least n steps, and (2) any parallel algorithm that finds its minimum needs at least $\log n$ parallel steps.

Without loss of generality, consider a cP system cell, in state S_1, where multiset A is given via functor a; e.g., multiset $A = \{1, 2, 2, 5\}$ is represented as $a(1)a(2)a(2)a(5)$. The following rulesets implement

various versions of a cP system minimum-finding algorithm. All these rulesets transit to state S_2 and construct a term with functor b, containing $\min A$. Some of these are destructive processes; if otherwise desired, one could first make a copy of the initial multiset A.

The following destructive ruleset is an emulation of the classical sequential minimum finding algorithm, which takes n steps:

$$
\begin{array}{lll}
S_1 \;\; a(X) \;\; \rightarrow_{\texttt{min}} \;\; S_2 \;\; b(X) & & \\
S_2 \;\; a(XY) \;\; b(X) \;\; \rightarrow_{\texttt{min}} \;\; S_2 \;\; b(X) & & a \geq b \\
S_2 \;\; a(X) \;\; b(X1Y) \;\; \rightarrow_{\texttt{min}} \;\; S_2 \;\; b(X) & & a < b
\end{array}
$$

The following destructive ruleset is an emulation of the classical parallel minimum finding algorithm, which takes $\log n$ steps. As long as there are more than one term a, the ruleset loops in state S_1, keeping minima between pairs. When only one a remains (containing the minimum value), the ruleset transits to state S_2 and tags the minimum.

$$
\begin{array}{l}
S_1 \;\; a(XY) \;\; a(X) \;\; \rightarrow_{\texttt{max}} \;\; S_1 \;\; a(X) \\
S_1 \;\; a(X) \;\; a(X1Y) \;\; \rightarrow_{\texttt{max}} \;\; S_1 \;\; a(X) \\
S_1 \;\; a(X) \;\; \rightarrow_{\texttt{min}} \;\; S_2 \;\; b(X)
\end{array}
$$

However, using the full associative power of cP systems, we can find a non-destructive version with two rules, which works in *just two steps* (regardless of the set cardinality). This is a substantial improvement over existing classical algorithms (both sequential and parallel). It starts by making a full copy of a as b, in one `max`-parallel step, and then deletes all non-minimal b values in another `max`-parallel step.

$$
\begin{array}{l}
S_1 \;\; \rightarrow_{\texttt{max}} \;\; S_1' \;\; b(X) \;\; | \;\; a(X) \\
S_1' \;\; b(X1Y) \;\; \rightarrow_{\texttt{max}} \;\; S_2 \;\; | \;\; a(X)
\end{array}
$$

Note that, if the minimum value appears several times in multiset A, then we will end with the same multiplicity of b's, each one containing the same value, $\min A$. If this is required, there are several ways to select only one copy and delete the rest – but we do not further deal with this issue here.

Moreover, using the full power of cP inhibitors (as logical negations, with local variables), we can even non-destructively solve the problem in just *one single step*, with one or two rules. This version is implemented by the following ruleset:

$$
\begin{array}{l}
S_1 \;\; \rightarrow_{\texttt{min}} \;\; S_2 \;\; b() \;\; | \;\; a() \\
S_1 \;\; \rightarrow_{\texttt{min}} \;\; S_2 \;\; b(1Z) \;\; | \;\; a(1Z) \;\; \neg \;\; (Z = XY) \;\; a(X)
\end{array}
$$

If A contains zero, then there is a term $a()$, and: (1) the first rule applies, constructing $b()$; (2) the second rule is not applicable. Otherwise (if there is no zero in A): (1) the first rule is not applicable; (2) the second rule constructs $b(1Z)$, a value which exists among a's, as $a(1Z)$, but there is NO other a containing a strictly lesser value, such as $a(X)$, where X is a sub-multiset of Z, $X \subseteq Z$. In the end, the newly constructed b will contain one copy of the minimum value of multiset A.

If multiset A does not contain zero values, i.e. $A \subseteq \mathbb{N}^+$, then the first rule can be safely omitted (as it will never be applicable). A similar ruleset can be devised for finding the maximum of a given set of natural numbers.

3. Description of our cP Systems Hamiltonian Path and Cycle Algorithms

We discuss in this section only digraphs, recalling that all undirected graphs can be represented quite simply as a directed graph with arcs in both directions simulating edges – this is in fact how we represent undirected graphs in our algorithm.

3.1. Hamiltonian Path

Our algorithm follows a simple approach, essentially a simple maximally parallel breadth-first search of the digraph. We start with a top-level cell enclosed by the skin membrane, and populated with subcells describing the problem digraph (recall that subcells in cP systems are the namesake complex objects - we hereafter use the term subcell in this paper). From there, a starting vertex of the problem digraph is randomly selected, and the top-level cell is then populated with the other initial required subcells. The evolution of the system then proceeds synchronously level-by-level through the different potential paths of the cycle by creating new subcells encoding the digraph traversals up to that point, expanding all possible paths from a given vertex which exclude any of the previously visited vertices.

Our algorithm requires a fixed set of **only four rules**, presented in Figure 1. Note that we do not define a class of rules and suggest that there should be instantiations of it to fit the specific problem, but instead we define precisely four rules that apply the same to all problems conforming to the basic assumptions described below.

At the beginning of the computation, we assume we have an elementary cell with the skin membrane, and that the set E of subcells of the form $E = \{e(f(i)\, t(j))\}_{i,j\in\mathbb{N};\ i\neq j}$ encoding the arcs of the problem digraph is already present inside the skin membrane. Subcell e represents an arc; f the origin vertex; and t the destination vertex. We further assume that the subcell $v(v(X), v(Y)...)$, listing the vertices of the problem digraph is present, though this could be derived from the subcells in E, if required. The system begins in state 1.

$$
\begin{array}{llll}
S_1 & v(v(R)Y) & \rightarrow_{\texttt{min}}\ S_2\ \ s(u(Y)\ p(h(R)p())) & (1)\\
S_2 & & \rightarrow_{\texttt{min}}\ S_3\ p'(P) & (2)\\
 & & \quad \mid s(u()\ p(P)) & \\
S_2 & & \rightarrow_{\texttt{max}}\ S_2\ s(u(Z)\ p(h(T)p(h(F)p(P)))) & (3)\\
 & & \quad \mid s(u(v(T)Z)\ p(h(F)p(P))) & \\
 & & \quad \mid e(f(F)\ t(T)) & \\
S_2 & s(\_) & \rightarrow_{\texttt{max}}\ S_2 & (4)
\end{array}
$$

Figure 1. Ruleset for our Hamiltonian Path Problem cP systems algorithm.

Rule (1) begins the computation by selecting an arbitrary vertex R from the subcell v to become the

starting point of the cycle, and creating our first s subcell. The s subcells represent steps down the tree of the exploration of the digraph (see Figure 5 later for an example), with the first step representing the selection of the starting vertex R. Each s subcell comprises two components: u, a set representing the remaining unexplored vertices in the digraph; and p which acts as a list as described above, and keeps track of the cycle's path so far. The R variable used in this rule represents the root of the computation, which is not used further in this algorithm, but becomes important for the HCP and TSP. The rule is applied in `min` mode, and the system transitions to state 2. Application of this rule takes one step.

Rule (2) is listed earlier despite being applied after rules (3) and (4), in order to enjoy an advantage in the weak priority ordering. After rules (3) and (4) have been applied enough times to leave the system only with s subcells that have no further nodes to explore in their u subcell, rule (2) will select and output a path p' chosen at random from amongst those deduced so far. This rule is applied in `min` mode, and application of the rule takes one step, with the system ending in state 3. The application of this rule represents the end of the evolution of the system.

Rule (3) is arguably the heart of the algorithm. So long as there are one or more vertex labels remaining in the unexplored vertex subcells u and a relevant e subcell available, this rule will be applied to each extant s subcell, and create new derivative s subcells that represent another step in the exploration of the digraph/another level in the exploration tree. The next selected vertex T for each instantiation will be removed from u and prepended to p. This rule is applied in `max` parallel mode, and the system remains in state 2. Application of this rule takes one step per remaining vertex in the digraph, or $n - 1$ steps in total.

Rule (4) runs in parallel with rule (3), and simply removes the extant s subcells from the system. Due to the parallel nature of cell-like P systems, where any number of rules can be applied concurrently so long as they do not conflict with each other, this rule can work in conjunction with rule (3) without issue, because changes to subcells are not performed until the end of the step. Note that neither rules (3) nor (4) can be applied alongside rule (2), because rule (2) changes the system's state, and therefore application of it conflicts with the latter two rules. Both later rules are applied at the same time, so that at the end of their application, the new s subcells have been created, and the pre-existing ones deleted. This rule is applied in `max` mode, and the system remains in state 2. Application of this rule takes one step per vertex in the digraph, or $n - 1$ steps in total – of note is that these steps are the same ones used for the application of rule (3), and occur simultaneously.

3.2. Hamiltonian Cycle

Finding a Hamiltonian cycle requires an expansion of the algorithm, specifically to record the initial starting vertex, and ensure that the paths produced also end there. The modified rules taking account of the changes are presented in Figure 2.

The major changes to this from the HPP algorithm are the addition of an r subcell to every s subcell, the change of rule (2) and the addition of a rule (5). The r subcell stores a vertex – the R selected in rule (1) – to keep track of the starting point of the cycle, where the cycle will also have to end. This is used by the new rule (2), which instead of outputting a randomly selected path, constructs new z subcells out of the s subcells. These new subcells contain the full path of the cycle, but are only instantiated if there is an e subcell representing an arc on the graph from the final vertex on the path back to the origin vertex R, by the action of the promoter. Finally, rule (5) selects at random from one of these new z subcells and outputs a final path subcell p'. Rule (5) runs in `min` mode. Application of this rule takes one step, and

$$
\begin{array}{llll}
S_1 & v(v(R)Y) & \rightarrow_{\texttt{min}} \; S_2 \;\; s(r(R)\; u(Y)\; p(h(R)p())) & (1) \\
S_2 & s(r(R)\; u()\; p(h(F)p(P))) & & \\
 & & \rightarrow_{\texttt{max}} \; S_3 \; z(p(h(R)p(h(F)p(P)))) & (2) \\
 & & \quad \mid e(f(F)\; t(R)) & \\
S_2 & & \rightarrow_{\texttt{max}} \; S_2 \; s(r(R)\; u(Z)\; p(h(T)p(h(F)p(P)))) & (3) \\
 & & \quad \mid s(r(R)\; u(v(T)Z)\; p(h(F)p(P))) & \\
 & & \quad \mid e(f(F)\; t(T)) & \\
S_2 & s(\_) & \rightarrow_{\texttt{max}} \; S_2 & (4) \\
S_3 & & \rightarrow_{\texttt{min}} \; S_4 \; p'(P) & (5) \\
 & & \quad \mid z(p(P)) &
\end{array}
$$

Figure 2. Ruleset for our Hamiltonian Cycle Problem cP systems algorithm.

the system ends in state 4, which represents the new termination point of the evolution of the system.

The time complexity of this algorithm can be summarised as Proposition 3.1:

Proposition 3.1. In total, the Hamiltonian cycle (path) algorithm takes $n + 3$ $(n + 1)$ operations, giving the algorithm a time complexity of $\mathcal{O}(n)$.

4. Our cP Systems Travelling Salesman Problem Algorithm

The expansion of the Hamiltonian Cycle algorithm to solve the Travelling Salesman Problem is fairly straightforward. The Travelling Salesman Problem is merely the Hamiltonian Cycle Problem with weights upon the vertices of the digraph, and the imposition of the additional constraint that the chosen Hamiltonian cycle is one with minimum total weight.

In presenting their algorithm to solve the TSP, some papers have assumed totally connected digraphs, and/or used an Euclidean distance metric to define the weight between two arcs. We however assume a digraph with pre-specified arc weights, which could be derived as a pre-processing step (using Euclidean distances if appropriate), though we also assume that all weights are strictly positive integers. Handling the case of negative weights is a remaining open problem. Furthermore, our algorithm works with digraphs of any level of density so long as at least one Hamiltonian path exists (and could be extended to appropriately handle the case where none exists).

Again the algorithm requires five rules, presented in Figure 3. The only differences between the rules in Figure 3 and Figure 2 are the addition of a weight/cost component to the e and s subcells, the tracking

of costs throughout the process, and the inhibitor added to rule (5), which ensures that the chosen path has minimum cost.

$$
\begin{array}{lll}
S_1 \quad v(v(R)Y) & \rightarrow_{\texttt{min}} \; S_2 \;\; s(r(R)\; u(Y)\; p(h(R)p())\; c(\lambda)) & (1) \\
S_2 \quad s(r(R)\; u()\; p(h(F)p(P))\; c(C)) & & \\
 & \rightarrow_{\texttt{max}} \; S_3 \; z(p(h(R)p(h(F)p(P)))\; c(CW)) & (2) \\
 & \qquad\qquad \mid e(f(F)\; t(R)\; c(W)) & \\
S_2 & \rightarrow_{\texttt{max}} \; S_2 \; s(r(R)\; u(Z)\; p(h(T)p(h(F)p(P)))\; c(CW)) & (3) \\
 & \qquad\qquad \mid s(r(R)\; u(v(T)Z)\; p(h(F)p(P))\; c(C)) & \\
 & \qquad\qquad \mid e(f(F)\; t(T)\; c(W)) & \\
S_2 \quad s(\_) & \rightarrow_{\texttt{max}} \; S_2 & (4) \\
S_3 & \rightarrow_{\texttt{min}} \; S_4 \; p'(P) \quad c'(1D) & (5) \\
 & \qquad\qquad \mid z(p(P)\; c(1D)) & \\
 & \qquad\qquad \neg\, (D = CW)\; z(p(\_)\; c(C)) &
\end{array}
$$

Figure 3. Ruleset for our Travelling Salesman Problem cP systems algorithm.

The e subcells store the cost of traversing an arc in the digraph with a new c subcell. The costs are extracted from the e subcells via pattern-matching, and accumulated in the s subcells through addition, as described in Section 2. Otherwise the first four rules are substantively unchanged from those presented in Figure 2. Rule (5), however, has been more notably modified, in that it now includes an inhibitor as a condition, as well as the promoter. The inhibitor acts to ensure that the path chosen is truly one of minimum cost, in accordance with the process described in Section 2.4.

The time complexity of this algorithm remains unchanged, as we merely expand the space complexity slightly. Thus the time complexity can still be summarised as Proposition 4.1:

Proposition 4.1. In total, the Travelling Salesman Problem algorithm takes $n + 3$ operations, giving the algorithm a time complexity of $\mathcal{O}(n)$.

5. Worked example

Consider a digraph G such as that in Figure 4. Ordinarily this would be shown as an undirected graph because all arcs are two-way, but it is presented as an equivalent directed graph so that it more closely matches the input digraphs as described for the cP systems rules described above, specifically with

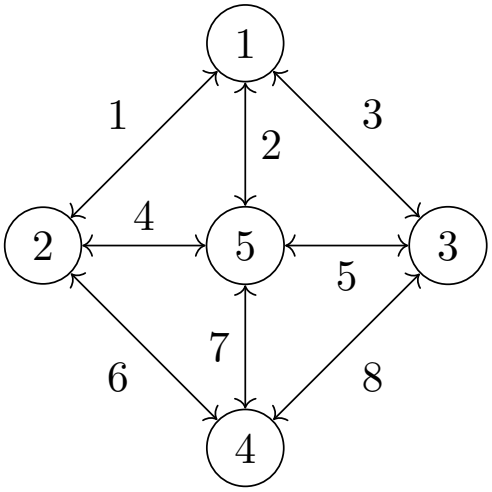

Figure 4. Sample weighted graph G with at least one Hamiltonian Cycle

regards to the set E of arc subcells. A quick examination will show that there is at least one Hamiltonian Cycle in this digraph, and thus there will be at least one cycle which has a minimum total weight. Figure 5 is a tree diagram showing the logical progression of the algorithm as applied to this digraph, assuming that vertex 1 is selected as the root of the Hamiltonian cycle. Vertices in bold are the ends of the paths with a minimum cost, while vertices in italics are the ends of the paths where there is no arc in the digraph such that a Hamiltonian cycle can be completed, based on the digraph traversal up to that point. The arcs are labelled with the cumulative weight of the path taken to reach the lower vertex.

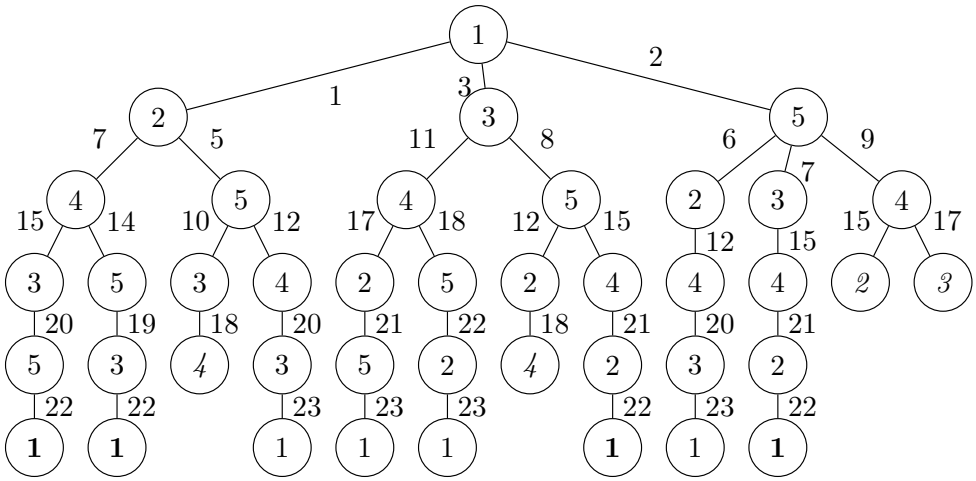

Figure 5. Tree diagram of the algorithm in action

The set of subcells contained inside the membrane at various points in the system's evolution are shown in Figures 6 to 9 (for the sake of legibility, when specifying the p subcells, Figure 9 adopts the compact presentation style for lists set out above in Section 2.3). The system starts with the subcells

$$e(f(1)\,t(2)\,w(1)) \quad e(f(1)\,t(3)\,w(3)) \quad e(f(1)\,t(5)\,w(2)) \quad e(f(2)\,t(1)\,w(1))$$
$$e(f(2)\,t(4)\,w(6)) \quad e(f(2)\,t(5)\,w(4)) \quad e(f(3)\,t(1)\,w(3)) \quad e(f(3)\,t(4)\,w(8))$$
$$e(f(3)\,t(5)\,w(5)) \quad e(f(4)\,t(2)\,w(6)) \quad e(f(4)\,t(3)\,w(8)) \quad e(f(4)\,t(5)\,w(7))$$
$$e(f(5)\,t(1)\,w(2)) \quad e(f(5)\,t(2)\,w(4)) \quad e(f(5)\,t(3)\,w(5)) \quad e(f(5)\,t(4)\,w(7))$$
$$v(v(1)\,v(2)\,v(3)\,v(4)\,v(5))$$

Figure 6. Set of subcells from G in the skin membrane at the initial state

shown in Figure 6, and the algorithm starts by applying rule (1), randomly selecting vertex 1 as the starting point of the Hamiltonian cycle and creating origin subcell $s(...)$ (full details of the contents of the subcells are provided in the figures), as shown in Figure 7.

$$e(f(1)\,t(2)\,w(1)) \quad \cdots \quad e(f(5)\,t(4)\,w(7))$$
$$s(r(1)\ \ u(v(2)\,v(3)\,v(4)\,v(5))\ \ p(h(1)p())\ \ c(0))$$

Figure 7. Set of subcells in the skin membrane after the application of rule one

Next, rule (3) is applied, creating the first level of subcells in the exploration tree. This creates new $s(r(R)\ \ u(...)\ \ p(h(...)p(h(1)p()))\ \ c(...))$ subcells, representing the potential paths of the cycle after one step. Rule (4) concurrently removes the old s subcells from the system. Figure 8 shows the subcells within the top-level cell at the end of this first application of rules (3) and (4).

$$e(f(1)\,t(2)\,w(1)) \quad \cdots \quad e(f(5)\,t(4)\,w(7))$$
$$s(r(1)\ \ u(v(3)\,v(4)\,v(5))\ \ p(h(2)p(h(1)p()))\ \ c(1))$$
$$s(r(1)\ \ u(v(2)\,v(4)\,v(5))\ \ p(h(3)p(h(1)p()))\ \ c(3))$$
$$s(r(1)\ \ u(v(2)\,v(3)\,v(4))\ \ p(h(5)p(h(1)p()))\ \ c(2))$$

Figure 8. Set of subcells in the skin membrane after a single application of rules three and four

Eventually, after repeating rules (3) and (4) five times, rule (2) will become applicable. At this point, rule (2) is applied, creating the z subcells that represent the final arc traversal from another vertex back

to the origin vertex, vertex 1. Finally, rule (5) selects one of those z subcells with minimum cost as the solution, and outputs the path and cost subcells relating to that cycle. Figure 9 outlines the subcells present in the system at this end point.

$$e(f(1)\,t(2)\,w(1)) \quad \cdots \quad e(f(5)\,t(4)\,w(7))$$
$$z(c(22)\ p(1|5|3|4|2|1))$$
$$\vdots$$
$$z(c(22)\ p(1|2|4|3|5|1))$$
$$c'(22) \quad p'(1|5|3|4|2|1)$$

Figure 9. Set of subcells in the skin membrane at completion of the computation, if rule (5) selects the subcell containing the path subcell representing the traversals 1 - 2 - 4 - 3 - 5 - 1.

To illustrate the progression of the algorithm through various branches of the exploration tree, consider the following examples, each beginning with the $s(...)$ and set of $e(...)$ subcells illustrated in Figure 7.

From $s(r(1)\ u(v(2)\,v(3)\,v(4)\,v(5))\ p(h(1)p())\ c(0))$, the subcell representing the beginning of the cycle at vertex 1, rule (3) will create, amongst others, an $s(r(1)\ u(v(3)\,v(4)\,v(5))\ p(h(2)p(h(1)p()))\ c(1))$ subcell representing an arc traversal to vertex 2 with a weight subcell $c(1)$. In turn, another new subcell, amongst others, will be derived from this subcell representing a further arc traversal to vertex 4, with a weight subcell of $c(7)$. This continues for subcells representing traversals to vertices 3 ($c(15)$) and 5 ($c(20)$), until finally the latter subcell contains an empty u subcell. For this subcell, rule (2) finds an e subcell that connects vertex 5 to R, the root vertex 1, and so creates a z subcell (the top z subcell in Figure 9) containing a $p(...)$ subcell representing the traversed path, and a subcell $c(22)$, representing the total cost of the cycle. This final subcell is potentially selected at random by rule (5), because 22 is the minimum cost possible in this particular digraph, when starting and finishing at vertex 1.

Conversely, another chain of subcell creations will occur as vertex 1 to vertex 5, with $c(2)$, vertex 5 to vertex 4 with $c(9)$, vertex 4 to vertex 2 with weight $c(15)$. At this point, $u(v(3))$ is non-empty, but there is no e subcell representing a transition from vertex 2 to vertex 3, so this subcell reaches a 'dead-end', and will be removed without further effect by rule (4).

Similarly, a progression will occur from vertex 1 to vertex 3 with $c(3)$, to vertex 5 with $c(8)$, to vertex 2 with $c(12)$, and to vertex 4 with $c(18)$. At this point, every vertex has been visited, and the subcell u in this particular s subcell is empty, but there is no e subcell representing a transition from the current vertex back to the origin, so no z subcell will be created based on it.

5.1. Directed graph example

While the above example is drawn as a directed graph, to better match with the specification of the edges in the set of subcells E, it was effectively an undirected graph. In order to demonstrate that our algorithm

is at least as effective in the directed case, we have slightly modified the graph shown in Figure 4, with the modified digraph presented in Figure 10 and the accompanying exploration tree in Figure 11. The updated set of subcells that are inside the skin membrane at the start of the computation are shown in Figure 12. The changes made are that the edges from 1 to 2 and from 5 to 3 have been removed, and the weights between 1 and 3 and 4 and 5 have been partially modified so that they are different in each direction.

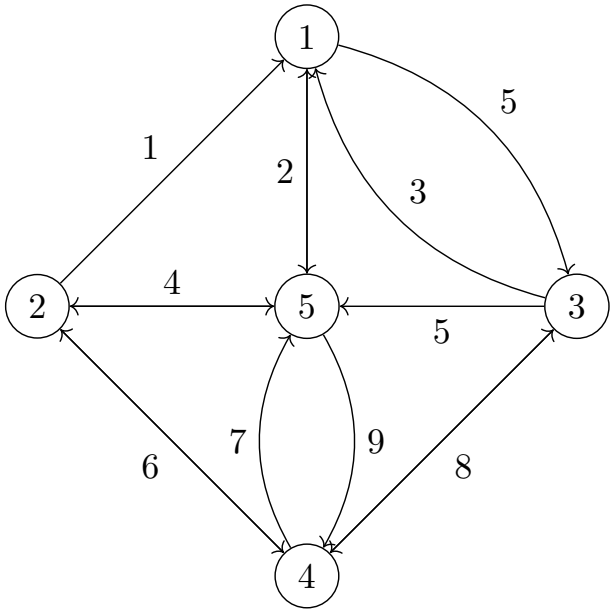

Figure 10. Sample weighted digraph H with at least one Hamiltonian Cycle

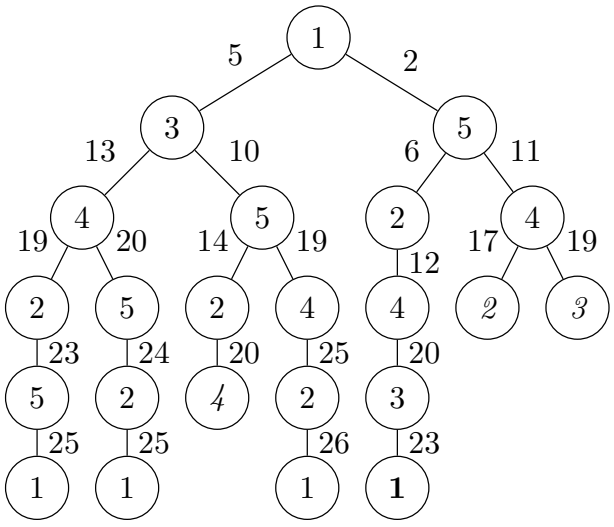

Figure 11. Tree diagram of the algorithm in action on the second, directed graph, H

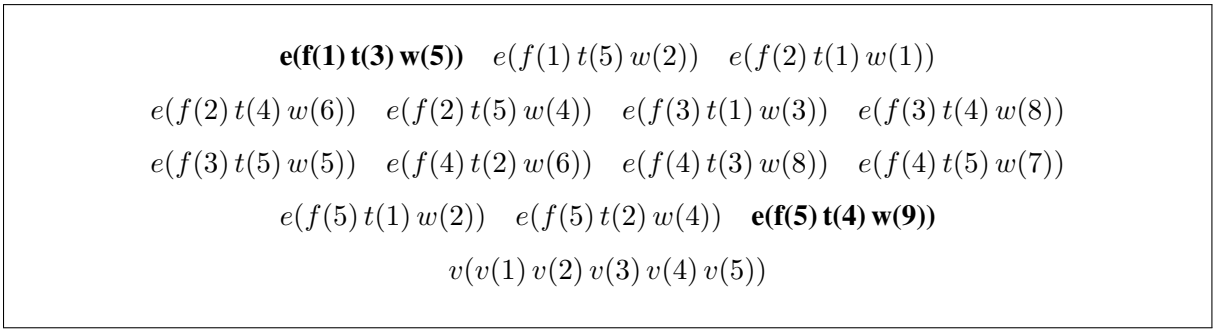

Figure 12. Set of subcells from H in the skin membrane at the initial state (those different from G are in bold)

These modifications have a significant effect upon the evolution of the system. There are fewer potential paths through the digraph, which results in the instantiation of fewer subcells, and so the exploration tree is consequently narrower. Thus, this particular application of the algorithm has a lower maximum space complexity. Note though that there is absolutely no change in the operation of the algorithm. The differences in the e subcells present at the start of the evolution of the system lead to a different result, without any change in the application of the rules.

6. Variations on the Algorithms

We have presented above algorithms to find a single Hamiltonian path or cycle, with minimum cost in the case of a weighted digraph (i.e., the TSP). With minimal modifications however, other results may be obtained from the algorithm, such as using a specific starting vertex or generating all possible paths/cycles.

To use a specific vertex as the starting point of the algorithm, rule (1) may be skipped by starting the system in state 2, with an s subcell containing the chosen vertex as the head of the p subcell supplied to the top-level cell.

For the HCP and TSP, rule (5) could be applied in `max` mode without any other changes so as to produce all possible Hamiltonian cycles (with minimum cost for the TSP) back to the starting node. This results in the output into the top-level cell of multiple p' subcells, to be handled further as appropriate. The same modification can be applied to rule (5) of the TSP algorithm to obtain all minimum-cost cycle paths. Likewise for each problem, if `max` mode is used on rule (1), then the paths/cycles starting at every vertex will be generated. These all potentially have the effect of increasing the space complexity of the system, which is not an issue for P systems with their infinite available space, but may impact software simulations.

In Section 4 we assumed that all arc weights are strictly positive integers. Weights of zero can be accommodated relatively easily. Should every possible cycle have a minimum weight of at least 1, then no changes are needed. If it is possible for there to be a cycle with a total weight of zero, then a sixth rule must be introduced, prior to the current rule (5), which in turn becomes rule (6). This rule simply detects a minimum of zero, as set out in Section 2.4. These new rules are shown in Figure 13 (recall that λ inside a subcell denotes the subcell as empty).

$$
\begin{array}{llll}
S_3 & \rightarrow_{\mathtt{min}} & S_4\; p'(P) \quad c'(\lambda) & (5) \\
 & & \mid z(p(P)\; c(\lambda)) & \\
S_3 & \rightarrow_{\mathtt{min}} & S_4\; p'(P) \quad c'(1D) & (6) \\
 & & \mid z(p(P)\; c(1D)) & \\
 & & \neg\; (D = CW)\; z(p(\_)\; c(C)) &
\end{array}
$$

Figure 13. Rules to find the minimum cost path in our TSP algorithm, when that path cost may be zero

7. Conclusion & Future Work

We have defined here a succinct cP systems algorithm for solving the Travelling Salesman Problem in $\mathcal{O}(n)$ time, by using the capacity of cP systems to create and manipulate complex subcells in only a few high-level steps. This algorithm builds on a simpler version for finding Hamiltonian paths and cycles, requires only a fixed set of five rules, and takes $n+3$ steps to find a solution for any connected digraph of size n, an improvement on the previous best known P systems-based solution to the TSP (see Table 1 for a comparison). Simple examples were provided to demonstrate the operation of the algorithm. The TSP algorithm can operate, without modification to the ruleset, on any arbitrary weighted graph for which there exists a Hamiltonian cycle. It requires only a specification of the graph encoded as subcells, and could be extended to detect the absence of a Hamiltonian cycle.

Table 1. Comparison of known exact P Systems solutions to the TSP

Algorithm	Num. of rules	Run time order
Guo & Dai [20]	∼50	$\mathcal{O}(n^2)$
Cooper & Nicolescu	5	$\mathcal{O}(n)$

Future developments for cP systems could proceed along a number of lines. As of yet, messaging between top-level cells has not been fully developed in cP systems, either using an asynchronous Actor-model-style, or a synchronous Communicating Sequential Processes style. Such techniques were not required here, but might prove very useful when creating any distributed algorithms.

One-way multiset unification occurs frequently in cP systems, with unification being used in every rule presented above. An efficient algorithm to perform this task would be highly beneficial for creating useful simulations of systems (we are not aware of an efficient algorithm in the case of multisets). For example, our simulations of the TSP algorithm written in functional programming languages (see Appendix A) regularly simply iterate over all relevant objects in the system, even though frequently most will be of little use in a given function call, and so the simulations could benefit from improved

unification in practice.

We would like to further develop the capacity to simulate cP systems, in particular developing more advanced techniques for translating cP systems rules to efficient parallel code. Work down this avenue has not begun as of yet however.

References

[1] Martín-Vide C, Păun G, Pazos J, Rodríguez-Patón A. Tissue P systems. Theoretical Computer Science. 2003;296(2):295–326. doi:10.1016/S0304-3975(02)00659-X.

[2] Pan L, Alhazov A. Solving HPP and SAT by P systems with active membranes and separation rules. Acta Informatica. 2006;43(2):131–145. doi:10.1007/s00236-006-0018-8.

[3] Song T, Wang X, Zheng H. Time-Free Solution to Hamilton Path Problems Using P Systems with d -Division. Journal of Applied Mathematics. 2013;2013:1–7. doi:10.1155/2013/975798.

[4] Chen H, He Z. A uniform solution to HPP in terms of membrane computing. In: International Conference on Artificial Intelligence and Computational Intelligence, AICI 2009. vol. 5855 LNAI. Shanghai, China: Springer Berlin Heidelberg; 2009. p. 59–68. doi:10.1007/978-3-642-05253-8_7.

[5] Tagawa H, Fujiwara A. Solving SAT and Hamiltonian cycle problem using asynchronous P systems. IEICE Transactions on Information and Systems. 2012;E95-D(3):746–754. doi:10.1587/transinf.E95.D.746.

[6] Xue J, Liu X. Solving directed Hamilton path problem in parallel by improved SN P system. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2013;7719 LNCS:689–696. doi:10.1007/978-3-642-37015-1_60.

[7] Jiménez MJP, Jiménez ÁR, Caparrini FS. Complexity Classes in Cellular Computing with Membranes. Natural Computing. 2003 sep;2(3):265–285. doi:10.1023/A:1025449224520.

[8] Mutyam M, Krithivasan K. P Systems with Membrane Creation: Universality and Efficiency. Machines, Computations, and Universality: Third International Conference,MCU 2001 Chişinău, Moldova, May 23-27, 2001, Proceedings. 2001;2055:276–287. doi:10.1007/3-540-45132-3_19.

[9] Smith SL, Imeson F. GLNS: An effective large neighborhood search heuristic for the Generalized Traveling Salesman Problem. Computers & Operations Research. 2017;87:1–19. doi:10.1016/j.cor.2017.05.010.

[10] Ezugwu AES, Adewumi AO. Discrete Symbiotic Organisms Search Algorithm for Travelling Salesman Problem. Expert Systems with Applications. 2017;doi:10.1016/j.eswa.2017.06.007.

[11] Cook WJ. In Pursuit of the Traveling Salesman; Mathematics at the Limits of Computation. Princeton University Press; 2012. Available from: `http://www.jstor.org/stable/j.ctt7t8kc`.

[12] Applegate DL, Bixby RE, Chvatl V, Cook WJ. The Traveling Salesman Problem; A Computational Study. Princeton University Press; 2006. Available from: `http://www.jstor.org/stable/j.ctt7s8xg`.

[13] Nishida TY. Membrane Algorithms. Membrane Computing: 6th International Workshop, WMC 2005, Vienna, Austria, July 18-21, 2005, Revised Selected and Invited Papers. Berlin, Heidelberg: Springer Berlin Heidelberg; 2006. p. 55–66. doi:10.1007/11603047_4.

[14] Manalastas P. Membrane Computing with Genetic Algorithm for the Travelling Salesman Problem. Theory and Practice of Computation: 2nd Workshop on Computation: Theory and Practice, Manila, The Philippines, September 2012, Proceedings. Tokyo: Springer Japan; 2013. p. 116–123. doi:10.1007/978-4-431-54436-4_9.

[15] He J, Xiao J, Shao Z. An adaptive membrane algorithm for solving combinatorial optimization problems. Acta Mathematica Scientia. 2014;34(5):1377–1394. doi:10.1016/S0252-9602(14)60090-4.

[16] Zhang G, Cheng J, Gheorghe M. A membrane-inspired approximate algorithm for traveling salesman problems. Romanian Journal of Information Science and Technology. 2011;14(1):3–19. Available from: https://www.researchgate.net/publication/264882106_A_membrane-inspired_approximate_algorithm_for_traveling_salesman_problems.

[17] Song X, Wang J. An Approximate Algorithm Combining P Systems and Active Evolutionary Algorithms for Traveling Salesman Problems. International Journal of Computers, Communications & Control. 2015;10(1):89–99. doi:10.15837/ijccc.2015.1.1567.

[18] He J, Zhang K. A Hybrid Distribution Algorithm Based on Membrane Computing for Solving the Multiobjective Multiple Traveling Salesman Problem. Fundamenta Informaticae. 2015;136(3):199–208. doi:10.3233/FI-2015-1151.

[19] Chen HZ, Lu JY, Wang YX. An approximate algorithm based on membrane computing for traveling salesman problems. Journal of Harbin Institute of Technology (New Series). 2011;18(SUPPL. 1):347–354.

[20] Guo P, Dai Y. A P System for Travelling Salesman Problem. In: Proceedings of the 18th International Conference on Membrane Computing (CMC18). International Membrane Computing Society; 2017. p. 147–165. Available from: http://computing.brad.ac.uk/cmc18/files/CMC18-Proceedings.pdf.

[21] Păun G. P Systems with Active Membranes: Attacking NP Complete Problems. Centre for Discrete Mathematics and Theoretical Computer Science, University of Auckland; 1999. Available from: https://www.cs.auckland.ac.nz/research/groups/CDMTCS/researchreports/.

[22] Păun G. 2. In: Păun G, editor. Prerequisites. Membrane Computing: An Introduction. Berlin, Heidelberg: Springer Berlin Heidelberg; 2002. p. 7–50. doi:10.1007/978-3-642-56196-2_2.

[23] Song B, Zhang C, Pan L. Tissue-like P systems with evolutional symport/antiport rules. Information Sciences. 2017 feb;378:177–193. doi:10.1016/j.ins.2016.10.046.

[24] Nicolescu R. Most Common Words A cP Systems Solution. In: Gheorghe M, Rozenberg G, Salomaa A, Zandron C, editors. Membrane Computing: 18th International Conference. Bradford, UK: Springer International Publishing; 2018. p. 214–229. doi:10.1007/978-3-319-73359-3.

[25] Păun G, Rozenberg G. One. In: Păun G, Rozenberg G, Salomaa A, editors. An Introduction to and an Overview of Membrane Computing. The Oxford Handbook of Membrane Computing. New York, NY, USA: Oxford University Press; 2009. p. 1–27.

[26] Nicolescu R, Ipate F, Wu H. Programming P Systems with Complex Objects. Membrane Computing: 14th International Conference, CMC 2013, Chişinău, Republic of Moldova, August 20-23, 2013, Revised Selected Papers. Berlin, Heidelberg: Springer Berlin Heidelberg; 2014. p. 280–300. doi:10.1007/978-3-642-54239-8_20.

[27] Nicolescu R, Wu H. Complex Objects for Complex Applications. Romanian Journal of Information Science and Technology. 2014;17(1):46–62.

A. Simulations

In order to demonstrate that this approach can be applied in practice to small problem graphs, sample simulations were written in SWI-Prolog, F# and Erlang for the TSP algorithm. In each case, the programs were written with an emphasis on matching the cP systems algorithm, rather than with a focus on memory or time efficiency. Better implementations from a real-world-use viewpoint could likely be created, but they may not reflect the cP systems rules quite as well. The Prolog program in particular matches very closely to the cP systems algorithm, and so is fully presented here. A complete program

listing for the Prolog program is in Listings 1 and 2 (Listing 1 defines the problem graph, and Listing 2 lists the program's rules corresponding to our algorithm), and a copy of the source code for each simulation is available at `https://github.com/jcoo092/cP-Systems-TSP`. While care was taken to keep these simulations similar to each other, differences between the languages inevitably means they are not identical.

All three languages are reasonably well-suited to implementing cP systems. As mentioned, the Prolog program in particular maps well to our algorithm, requiring only 7 Prolog rules in total, plus a variable number of facts specifying the problem graph, 17 in this case. The functional elements of F# such as higher-order functions also allow a reasonable approximation, if perhaps not with quite the fidelity of Prolog. Erlang, possibly owing to the fact it was originally implemented atop Prolog, appears to fall in between the two approaches, leaning more towards the functional side. It is clear from these programs that there is at least one potential close mapping from cP systems to both logic and functional programming languages. We note however that the emphasis here was on demonstrating the similarities of the languages to cP systems, and not on performance. As such, no attempt has yet been made to optimise the simulations.

To gauge their 'real-world' effectiveness, the simulations were informally tested with increasing digraph sizes. All three ably coped with digraph sizes of up to 10 vertices, returning an answer at most in a matter of a few seconds. F# and Erlang struggled somewhat at 11 vertices, with the latter taking more than one minute to complete, while Prolog quickly failed with an out-of-stack-memory error. The F# simulation was tested on a digraph of size 12, but the test was terminated after 90 minutes running time due to time constraints, before it completed and returned a minimum cost path. Due to this, a 12 vertex run was not attempted with Erlang.

Considering that for a totally-connected 11 vertex digraph, at the 11th step almost 40 million (11!) subcells would be required, it is unsurprising that memory limits may become an issue. These results suggest that, while the fundamental process does indeed lead to determining lowest-cost paths, much more effective use of memory will be required in order to make software simulations practically relevant with highly connected digraphs of any significant size (i.e. greater than approximately 11 or 12 vertices).

A comparison of Figures 5 and 11 suggests, however, that for graphs that are not totally connected, the implementations may cope much better, as would be expected. With ultimately fewer paths to explore, the explosion in the number of objects will not be so great, and thus the space requirements will be lower. This is not unique by any means to the cP systems version of the TSP - any method to solve the TSP which involves a breadth-first search of the graph, taking into account whether nodes have already been visited, will experience the same.

A.1. Prolog simulation

It is perhaps interesting to contrast the Prolog method for exploring the problem space and finding an answer with that of cP systems. On the surface they appear quite similar – a handful of rules and some initial statements specific to the problem, combined with unification. In some ways, however, they appear also to be opposing duals of each other. Prolog in general, and thus in the case of our program, works on a top-down backward-chaining approach, whilst cP systems works on a bottom-up forward-chaining approach.

That is to say that (sequential) Prolog tries to do the least work and use the least space possible, so it starts with the definition of the requested inference (an invocation of `go` in our program), and only

evaluates elements of that definition as it discovers they are needed in order to provide a result. In the case of the digraph exploration, this essentially means that it tries to perform a linear depth-first search of the digraph, stopping as soon as it has an answer – though with the TSP it inevitably must evaluate every cycle in order to know which has minimum cost.

Conversely, our cP systems algorithm starts with the definition of the problem graph, and does all possible work in order to find the result in what is essentially a breadth-first search. All possible cycles from the root node are instantiated and eventually checked for minimum cost. The ability of cP systems to perform these instantiations and the comparison completely in parallel means that a relatively small number of steps are required to find the minimum – though at the cost of a potentially (exponentially) large space and processing complexity.

```
e(1, 2, 1).  e(1, 3, 3).  e(1, 5, 2).  e(2, 1, 1).  e(2, 4, 6).  e(2, 5, 4).
e(3, 1, 3).  e(3, 4, 8).  e(3, 5, 5).  e(4, 2, 6).  e(4, 3, 8).  e(4, 5, 7).
e(5, 1, 2).  e(5, 2, 4).  e(5, 3, 5).  e(5, 4, 7).  v([1, 2, 3, 4, 5]).
```

Listing 1: SWI-Prolog code defining the example problem undirected graph G shown in Figure 4

```
s(R, [], [F| P], C, Ph, Ch) :- e(F, R, W), CW is C + W,
                                Ph = [R, F| P], Ch = CW.

s(R, Y, [F| P], C, Ph, Ch) :- member(T, Y), delete(Y, T, Z), e(F, T, W),
                         CW is C + W, s(R, Z, [T, F| P], CW, Ph, Ch).

h(R, Y, H) :- findall(z(Ph,Ch), s(R, Y,[R],0,Ph,Ch), H).

minh([z(P1,C1)], [z(P1,C1)]).
minh([z(P1,C1), z(_P2,C2)| H], M) :- C1 =< C2, !, minh([z(P1,C1)| H], M).
minh([z(_P1,_C1), z(P2,C2)| H], M) :- minh([z(P2,C2)| H], M).

go(M) :- v(X), member(R, X), delete(X, R, Y), !, h(R, Y, H), minh(H, M).
```

Listing 2: Complete SWI-Prolog code for the rules of our TSP algorithm