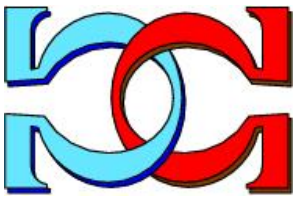
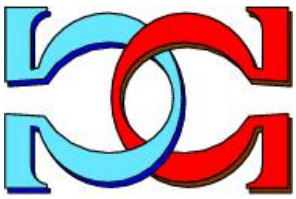
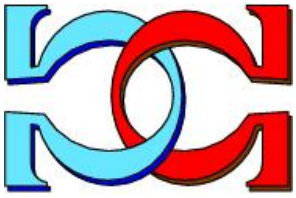
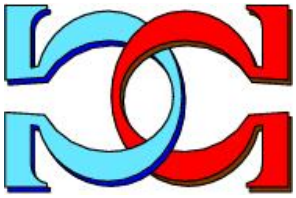
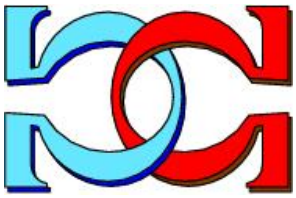
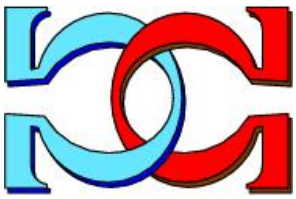
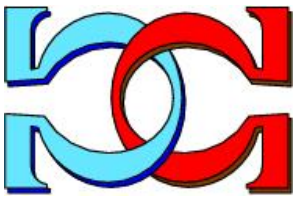

CDMTCS Research Report Series

The Travelling Salesman Problem in cP Systems

James Cooper and Radu Nicolescu
Department of Computer Science,
University of Auckland,
Auckland, New Zealand

CDMTCS-510
August 2017

Centre for Discrete Mathematics and
Theoretical Computer Science

The Travelling Salesman Problem in cP Systems

James Cooper and Radu Nicolescu
Department of Computer Science
The University of Auckland, Private Bag 92019
Auckland, New Zealand
jcoo092@aucklanduni.ac.nz
r.nicolescu@auckland.ac.nz

Abstract

The Travelling Salesman Problem (TSP) is a long-standing and well-known NP-hard problem, concerned with computing the lowest cost Hamiltonian cycle on a weighted graph. Many solutions to the problem exist, including some from the perspective of P Systems, almost all of which have combined membrane computing with other approaches for approximate solution algorithms. A recent paper presented a brute-force style P Systems solution to the TSP, exploiting the ability of P Systems to reduce time complexity in exchange for space complexity, but the resultant system had a relatively high number of rules. Inspired by this paper, and seeking a more concise representation of a brute-force TSP algorithm, we have devised a P Systems algorithm based on cP Systems which requires five rules and takes n + 3 steps. This paper describes said algorithm and an example use of it, and summarises simulations of the system written in Prolog and F#.
Keywords: Graph theory, Hamiltonian cycles, Travelling Salesman Problem, P Systems, cP Systems.

1 Introduction

The Travelling Salesman Problem (TSP) is a long-standing, well-known computationally (NP-) hard problem in Computer Science and related fields. Briefly, the problem is about finding the minimum cost Hamiltonian cycle in a weighted graph, where a Hamiltonian cycle is a traversal of the graph that ends where it began, but visits every node exactly once after the start. 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 has been studied extensively, spawning many papers, dissertations and books on the topic (e.g. [16, 4, 3, 1] amongst many, many others), and many sophisticated algorithms have been developed to efficiently solve the problem, 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 P Systems perspective.

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 effectively to search a space of solutions to the TSP for a given graph, for an approximate solution (note however the earlier paper [8], which gave a quadratic time solution to related Hamiltonian Path Problems, and more generally demonstrated the capacity of P Systems to trade time complexity for space complexity). Others built on this approach by integrating techniques such as Genetic Algorithms [9, 6], Ant Colony Optimisation [18] and Active Evolution [17], along with more complex approaches for multiple salesman problems [7]. A paper by Chen *et al.* [2] was apparently also written on the topic, but no copy of that paper could be located.

All these papers however have been written from the perspective of approximate solutions to the TSP, generally taking 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 object rules. More recently however, Guo and Dai published a paper on solving the TSP in the exact case using P Systems [5]. By exploiting the property of P Systems that time complexity can often be exchanged for space complexity [14, 8], the authors derived a P Systems algorithm that can solve the TSP in $\mathcal{O}(n^2)$ time.

Inspired by [5], we seek to derive a cP Systems algorithm for solving the TSP to exploit fully the power of P Systems' theoretical infinite space, as well as the compactness of representation of cP Systems. Such systems have been described extensively in prior papers, in particular [12, 10, 11], although, unlike in [10], for the sake of simplicity we do not use the Actor model in our functional programming simulation for this paper as it would be 'overkill'. Use of the Actor model is a likely approach to take if seeking to extend the algorithm to work in the distributed case though.

For the sake of space, we hereafter assume that the reader is familiar with the basic concepts of P Systems (see [15] for a good, if slightly old, introduction to P Systems), cP Systems and the TSP. Our algorithm, and in particular the rules for it, are presented in Section 2. We provide a worked example, applying our algorithm to a specific weighted digraph, in Section 3, and discuss practical simulations written in SWI-Prolog and F# in Section 4, the code for which is given in Appendix B. Finally, we conclude the paper and suggest some further possible directions for future work in Section 5.

2 Description of our brute-force TSP algorithm

The algorithm follows a simple approach, essentially a simple maximally parallel breadth-first search of the graph. We start with an elementary cell enclosed by the skin membrane, and populated with objects describing the problem graph. From there, a starting node of the problem graph is randomly selected, and populated with the other initial objects required. The computation then synchronously steps through the different potential paths of the cycle by creating new objects encoding the graph traversal up to that point, expanding all possible paths from a given node which exclude any of the previously visited nodes.

If this process were to be thought of as a tree, then each step would be constructing the next level of the tree. Once all possible full explorations of the graph have been generated, new objects are generated for those ending nodes which have a link back to the starting node. A final step selects one of the minimum cost paths at random.

In presenting their algorithm to solve the TSP, some papers have assumed totally connected graphs, and/or used an Euclidean distance metric to define the weight between two arcs. We however assume a graph with pre-specified arc weights, which could be derived as a pre-processing step (using Euclidean distances if appropriate). Furthermore, our algorithm works with graphs of any level of density so long as at least one Hamiltonian cycle exists (and could be extended to appropriately handle the case where none exists), and works for both directed and undirected graphs.

At the beginning of the computation, we assume we have an elementary cell with the skin membrane, and that the set E of objects of the form

$$E = \{e(f(i)\, t(j)\, w(k))\}_{i,j,k \in N;\ i \neq j}$$

encoding the arcs of the problem graph, is already present inside the skin membrane. Object $e()$ represents an arc, $f()$ the origin node, $t()$ the destination node, and $w()$ the weight of the arc. We further assume that the object $v(v(X), v(Y)...)$, listing the vertices of the problem graph is present, though this could be derived from the objects in E, if required. We further assume that there is an object $n(N)$ present, where $N = |v|$ (i.e. the number of nodes in the graph), though this could be derived by counting the objects present in v. The system begins in state 1.

The algorithm requires **only five rules**, presented in Figure 1. These rules are presented in weak-priority order, and are explained below.

Rule (1) begins the computation by selecting an arbitrary node from the object v to become the starting point of the cycle (encoded as $r(R)$), and creating our first s object. The s objects represents steps along the graph, with the first step representing the selection of the node. Each s object comprises four components, r; u, a set representing the remaining unexplored nodes in the graph; p which keeps

$$
\begin{array}{lll}
S_1 \quad v(v(R)Y) & \rightarrow_{\texttt{min}} \; S_2 \;\; s(r(R)\; u(Y)\; p(h(R)p())\; c(0)) & (1) \\
S_2 \quad s(r(R)\; u()\; p(h(F)p(P))\; c(C)) & & \\
& \rightarrow_{\texttt{max}} \; S_3 \; z(p(h(R)p(h(F)p(P)))\; c(CW)) & (2) \\
& \qquad\qquad \mid e(f(F)\; t(R)\; c(W)) & \\
S_2 & \rightarrow_{\texttt{max}} \; S_2 \; s(r(R)\; u(Z)\; p(h(T)p(h(F)p(P)))\; c(CW)) & (3) \\
& \qquad\qquad \mid s(r(R)\; u(v(T)Z)\; p(h(F)p(P))\; c(C)) & \\
& \qquad\qquad \mid e(f(F)\; t(T)\; c(W)) & \\
S_2 \quad s(\_) & \rightarrow_{\texttt{max}} \; S_2 & (4) \\
S_3 & \rightarrow_{\texttt{min}} \; S_4 \; p'(P) \quad c'(C1W) & (5) \\
& \qquad\qquad \mid z(p(P)\; c(C1W)) & \\
& \qquad\qquad \neg\; z(p(\_)\; c(C)) &
\end{array}
$$

Figure 1: Ruleset for our Travelling Salesman Problem cP Systems algorithm.

track of the cycle's path so far; and c which tracks the cost so far of the cycle. The rule is applied in min mode, and the system transitions to state 2. Application of this rule takes one step.

The object p acts akin to a classic singly-linked list, recursively preserving inside it a head element h, as well another p object as the remainder/tail of the path list, which may be empty, as in this rule. Thus, for the initial s object, it is a list of head R (the root node) cons the empty list.

Rule (2) is listed earlier despite being applied after rules (3) and (4), in order to enjoy an advantage in weak priority ordering. It takes the final set of s objects, which have reached the point of exploring the entire graph and thus contain empty u objects, and extracts from them z objects that each keep p and c objects taken from the s objects, which will be used in the application of the final rule. These z objects are only created when there is an e object describing an arc in the graph from the most recently explored node (the head of the path list) back to the root node, i.e. only when there is a possibility of a Hamiltonian cycle in the graph based on the path traversed so far. This rule simultaneously removes the final s objects, keeping the working space of our system relatively clean. The rule is applied in max mode, and the system transitions to state 3. Application of this rule takes one step.

Rule (3) is arguably the heart of the algorithm. So long as there are one or more node labels remaining in the unexplored node objects u and a relevant e object available, this rule will be applied to each extant s object, and create new derivative s objects that represent another step in the exploration of the graph/another level in the exploration tree. The next selected node for each instantiation will be removed from u and prepended to p, with the associated weight of the arc added to c. This rule is applied in max mode, and the system remains in state 2. Application of this rule takes one step per node in the graph, or n steps in total.

Rule (4) runs in parallel with rule (3), and simply removes the extant s objects from the system. Due to the parallel nature of 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 objects 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 objects 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 node in the graph, or n steps in total - of note is that these steps are the same ones used for the application of rule (3), and occur simultaneously.

Finally, rule (5) non-deterministically selects a z object with minimum cost by choosing an object such that there is no other object with a lower cost, and outputs the cost and path objects of that z object, completing the computation. This takes one step.

The time complexity of this algorithm can be summarised as Proposition 1:

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

For space complexity however, in the worst case of a totally connected graph, at the last application of rules (3) and (4), a total of (n-1)! s objects will be created, and an equal number of z objects with rule (2). Under the P Systems model of infinite space this is no issue, but of course can become a problem for real-world simulations.

2.1 Rule for selection of minimum weight object

Rule (5), the rule for arbitrarily selecting one of the minimum weight z objects, is perhaps unusual enough to warrant further attention. The rule creates one each of a c' object encoding the final minimum cost, and a p' object which contains the total final path, after non-deterministically choosing one of the z objects with minimum cost. The core of the functioning of the rule is the interaction between the promoter and inhibitor. The inhibitor states that there cannot be an object with cost C, while the promoter requires an object with at least weight $C1$ (i.e C + 1) - the W, like any numeric variable, can potentially be zero. The combined effect of this is to select an object for which there is no object with a lesser cost.

It may be tempting to think that, because the variables C and W can potentially take any number, it would be possible to select them such that one does not choose the minimum-cost object. What if one selects C as the minimum cost in any object, but W as any arbitrary natural number? Then it would appear that the rule would select a greater-than-minimum cost object, as the rule would seek out an object with at least $C + 1$ cost. Such a choice however violates the constraints of the rule. The inhibitor effectively states for the object selected, there must be no object with cost C such that C is less than the selected object's total cost, because if there is, then the number assigned to C can be modified from its initially set value to a lesser number equal to the lower cost, and the rule is then violated by there existing an object with C less than $C1W$.

As an example, consider the situation where there are objects

$$c_a(3) \quad c_b(4) \quad c_c(4) \quad c_d(5) \quad c_e(6)$$

. Clearly, the minimum cost object is c_a. What if one were to determine $C = 3$ and $W = 0$? Then the rule would select c_b or c_c as the minimum cost object. That clearly violates the inhibitor specification however, because it is possible to set $C = 2$ and find an object that fits to such a selection. It seems that the 'correct' selection in this case is $C = 2$ and $W = 0$. Other selections would not conform to the rule, as it would be possible to subtract from C and add to W and arrive at a valid object, meaning that there exists an object which could be determined to have cost C less than the selected object, and therefore the selection of the greater C is not valid according to the rule.

3 Worked example

Consider a graph G such as that in Figure 2. 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 graph as described for the cP Systems rules described above, specifically for the set E of arc objects. Quite obviously, there is at least one Hamiltonian Cycle in this graph, and 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 graph. Nodes in blue are the ends of the paths with a minimum cost, while nodes in red are the ends of the paths where there is no arc in the graph such that a Hamiltonian Cycle can be completed, based on the graph traversal so far. The arcs are labelled with the cumulative weight of the path taken to reach the lower node.

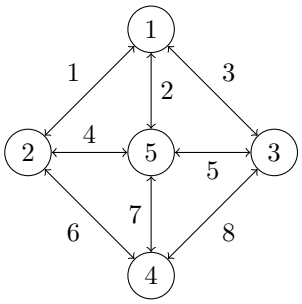

Figure 2: Sample weighted graph G with at least one Hamiltonian Cycle

$$
\begin{array}{cccc}
e(f(1)\,t(2)\,w(1)) & e(f(1)\,t(3)\,w(3)) & e(f(1)\,t(5)\,w(2)) & e(f(2)\,t(1)\,w(1)) \\
e(f(2)\,t(4)\,w(6)) & e(f(2)\,t(5)\,w(4)) & e(f(3)\,t(1)\,w(3)) & e(f(3)\,t(4)\,w(8)) \\
e(f(3)\,t(5)\,w(5)) & e(f(4)\,t(2)\,w(6)) & e(f(4)\,t(3)\,w(8)) & e(f(4)\,t(5)\,w(7)) \\
e(f(5)\,t(1)\,w(2)) & e(f(5)\,t(2)\,w(4)) & e(f(5)\,t(3)\,w(5)) & e(f(5)\,t(4)\,w(7)) \\
\multicolumn{4}{c}{v(v(1)\,v(2)\,v(3)\,v(4)\,v(5)) \quad n(5)}
\end{array}
$$

Figure 3: Set of objects from G in the skin membrane at the initial state

The set of objects contained inside the membrane at various points in the computation are shown in Figures 3, 4, 6 and 7. The algorithm starts by applying rule (1), selecting node 1 as the starting point of the Hamiltonian cycle, creating origin object $s(...)$ (full details of the contents of the objects are provided in the figures). Should a particular node be preferred as the starting node, the object created by created by rule (1) could instead by supplied from the outset, and rule (1) removed from the rules list, with the system beginning at state 2.

Next, rule (3) is applied, creating the first level of objects in the exploration tree (Figure 5). This creates new $s(r(R)\,u(...)\,p(h(v(1))p(...))\,c(...))$ objects, representing the potential paths of the cycle after one step. Rule (4) concurrently removes the old s objects from the system.

Eventually, after repeating rules (3) and (4) five times, rule (2) will become applicable. At this point, rule (2) is applied, creating the z objects that represent the final arc traversal from another node back to the origin node, node 1. Finally, rule (5) selects one of those z objects with minimum cost as the solution, and outputs the path and cost objects relating to that cycle.

For example, from the object representing node 1, rule (3) will create, amongst others, an object representing an arc traversal to node 2 with a weight object $c(1)$. In turn, another new object, amongst others, will be derived from this object representing a further arc traversal to node 4, with a weight object of $c(7)$. This continues for objects representing traversals to nodes 3 ($c(15)$) and 5 ($c(20)$), until

$$
\begin{array}{c}
e(f(1)\,t(2)\,w(1)) \quad \cdots \quad e(f(5)\,t(4)\,w(7)) \\
v(v(1)\,v(2)\,v(3)\,v(4)\,v(5)) \quad n(5) \\
s(r(R)\,u(v(2)\,v(3)\,v(4)\,v(5))\,p(h(v(1))p())\,c(0))
\end{array}
$$

Figure 4: Set of objects in the skin membrane after the application of rule one

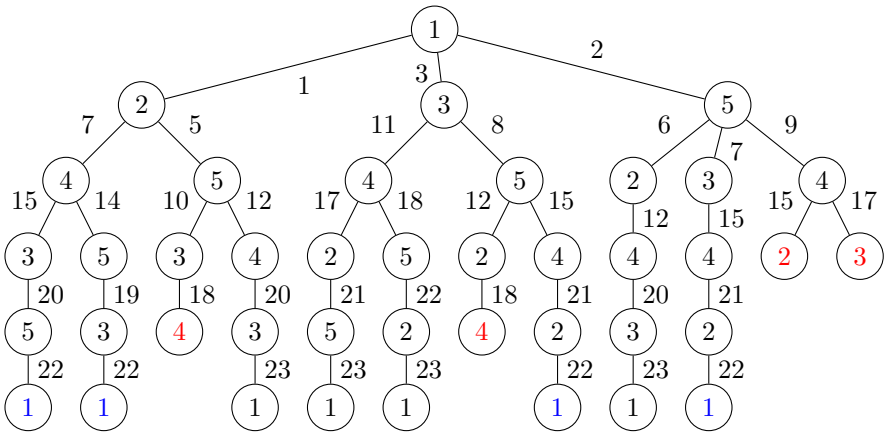

Figure 5: Tree diagram of the algorithm in action

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

Figure 6: Set of objects in the skin membrane after a single application of rules three and four

finally the latter object contains an empty u object. For this object, rule (2) finds an e object that connects node 5 to R, the root node 1, and so creates a z object containing a p object representing the traversed path, and an object $c(22)$, representing the total cost of the cycle. This final object is potentially selected at random by rule (5).

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

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

4 Simulations

In order to demonstrate that this approach works in practice, small programs were written in SWI-Prolog and F# to solve the example problem presented above. In both instances, the programs were written with an emphasis on matching the cP Systems rules, 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 as well. The complete program listings are in Appendix B, and a copy of the source code is available at `https://github.com/jcoo092/cP-Systems-TSP`.

$$e(f(1)\,t(2)\,w(1)) \quad \cdots \quad e(f(5)\,t(4)\,w(7))$$
$$v(v(1)\,v(2)\,v(3)\,v(4)\,v(5)) \quad n(5)$$
$$z(c(22)\,p(h(1)p(h(v(5))p(h(v(3))p(p(h(v(4))p(p(h(v(2))p(p(h(v(1))p())))))))))$$
$$\vdots$$
$$z(c(22)\,p(h(1)p(h(v(2))p(h(v(4))p(p(h(v(3))p(p(h(v(5))p(p(h(v(1))p())))))))))$$
$$c(22) \quad p(h(1)p(h(v(5))p(h(v(3))p(p(h(v(4))p(p(h(v(2))p(p(h(v(1))p()))))))))$$

Figure 7: Set of objects in the skin membrane at completion of the computation, if rule (5) selects the object containing the path object representing the traversals 1 - 2 - 4 - 3 - 5 - 1.

4.1 Prolog simulation

This simulation is very small, and almost maps directly to the cP Systems description. It requires only 30 lines of code, more than half of which (lines 1 - 18) are the instantiation of the initial objects of the problem as Prolog facts. Rules (1) - (3) are handled by three Prolog rules, while rule (4) is not considered in this implementation. Line 20 matches rule (1), line 23 matches rule (3), and line 24 matches rule (2). The only significant variation from the cP Systems algorithm is the implementation of Rule 5, which is implemented using four other rules, as a recursive search across the list of all possible Hamiltonian cycles in the graph for one with minimum cost.

Despite the close approximation of the rules with the Prolog program, there are some notable differences in operation. Importantly, the model of exploration is fundamentally different. The cP Systems algorithm essentially explores the graph in a maximally parallel forward-only breadth-first search using object rewriting rules on extant terms. Whereas instead the Prolog program works as a sequential[1] depth-first exploration of the graph with backtracking, using logical inference rules between virtual goals.

This significant change in search style means that we no longer trade space complexity for time complexity, returning the Prolog implementation to a $\mathcal{O}(n!)$ running time, but enabling much larger graphs to be explored gradually. Another notable difference is that the cP Systems s objects use an object u containing a set of the unvisited nodes to keep track of what nodes are available, whereas our Prolog simulation uses a list to represent u.

At small graph sizes this is largely irrelevant, but it does mean that in the worst case, assessing whether an arc is relevant to the current search takes $\mathcal{O}(n)$ steps, which could be relatively slow in the case of a large set. Using a more natural set representation, such as one based on an associative container using hash functions, would bring the implementation closer to our algorithm and make the lookup of the available nodes a constant time operation.

4.2 F# simulation

The small ruleset for this algorithm maps well to F#, and the entire example takes fewer than 50 lines of code.

Lines 1 to 12 of the sample code correspond to the creation of the initial objects present in the system prior to the beginning of our algorithm and the definition of relevant types. Lines 35 and 36, and the call to hgoal, represent rule (1). Rule (2) is represented by the conditional statement in sgoal and accompanying call to sgoal. Rule (3), and it's repeated application, is represented by the visit function, and the calls to it on the else branch sgoal's conditional. Rule (4) is implicitly represented by the tail recursion of the sgoal function. When the next function call is made, the current slist variable goes out

[1]It would be preferable to use a parallel implementation insofar as possible, but no current inherently parallel implementation of Prolog could be easily located.

of scope, and thus the memory allocated to the variable is freed by the garbage collector. Finally, rule (5) is represented by the minh function.

It is the case that this program involves a lot of instantiations/memory allocations, with relatively few calculations performed. It could arguably be seen as memoization taken to an extreme. This may in fact limit its timely operation in real life, considering that memory allocation is a relatively slow operation. Moreover, the approach of this algorithm is to expand the memory used as required, which in the worst case will grow at a factorial rate, limiting the real-world utility of the program.

Given that, for a fully connected graph, at the eleventh step of instantiations almost 40 million (**11!**) objects would be created, it is no surprise. Assuming that each object requires 100 bytes of memory, roughly 3.7 GiB of memory would be needed to hold the objects for only the eleventh step, meaning that in total during the operation of the build function for the eleventh step, more space would be required in memory than the 4 GiB limit of 32-bit systems allows.

5 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 objects in only a few high-level steps. This algorithm requires only five rules, and takes n + 3 steps to find a solution. A simple example was provided to demonstrate the operation of the algorithm. This algorithm can operate on both directed and undirected graphs for which there exists a Hamiltonian cycle, and could be extended to detect the absence of one.

It is a trivial task to create a basic simulation of this algorithm in a functional language such as F#, although due to the factorial-scale increase in the number of objects created, a naïve implementation may fail through lack of memory on graphs larger than 10 nodes. Another simulation was created in a logic programming language, specifically SWI-Prolog, which in many ways closely fits with the algorithm presented, but operates in a purely sequential manner without creating extra objects, thus returning to a time complexity of $\mathcal{O}(n!)$.

5.1 Future work

There are multiple potential directions that further work could take, primarily with regards to the practical implementation. Given the algorithm's reliance on memory space, finding ways to decrease significantly the memory used could lead to a greater ability to apply the algorithm to larger problems, possibly by trading some memory allocations for more calculations.

Given that memory allocation can be a relatively slow operation as compared to most calculations performed on a CPU, this may in fact improve performance overall, while also increasing the size of graphs that can be considered. This last approach, if carried out appropriately, might make the algorithm amenable to running on a GPU - although given that it currently involves a significant number of semi-random accesses to memory, this may not be feasible. Also, further exploration of the parallelism opportunities available in modern Prolog implementations may yield a faster algorithm without overly increasing the required memory.

Alternatively, the algorithm could be restructured to resemble more closely tissue-like P Systems, which are well-suited to implementations with the Actor model, which are in turn well-suited to running on the Cloud, where access to potentially much larger memory capacities may enable larger graphs to be considered. Other theoretical P Systems models might also be considered. Given Spiking Neural P Systems' focus on the communication of numbers/weights between neurons, the model may be a good fit to this problem.

References

[1] Applegate, D.L., Bixby, R.E., Chvatl, V., Cook, W.J.: The Traveling Salesman Problem; A Computational Study. Princeton University Press (2006), `http://www.jstor.org/stable/j.ctt7s8xg`

[2] Chen, H.Z., Lu, J.Y., Wang, Y.X.: An approximate algorithm based on membrane computing for traveling salesman problems. Journal of Harbin Institute of Technology (New Series) 18(SUPPL. 1), 347–354 (2011)

[3] Cook, W.J.: In Pursuit of the Traveling Salesman; Mathematics at the Limits of Computation. Princeton University Press (2012), http://www.jstor.org/stable/j.ctt7t8kc

[4] Ezugwu, A.E.S., Adewumi, A.O.: Discrete symbiotic organisms search algorithm for travelling salesman problem. Expert Systems with Applications (2017), http://www.sciencedirect.com/science/article/pii/S0957417417304141

[5] Guo, P., Dai, Y.: A p system for travelling salesman problem. In: Proceedings of the 18th International Conference on Membrane Computing (CMC18). pp. 147–165. Springer-Verlag (24-28 July 2017)

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

[7] He, J., Zhang, K.: A hybrid distribution algorithm based on membrane computing for solving the multiobjective multiple traveling salesman problem. Fundamenta Informaticae 136(3), 199–208 (2015)

[8] Jiménez, M.J., Péerez, Jiménez, Á.R., Caparrini, F.S.: Complexity classes in models of cellular computing with membranes. Natural Computing 2(3), 265–285 (2003), http://dx.doi.org/10.1023/A:1025449224520

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

[10] Nicolescu, R.: Parallel Thinning with Complex Objects and Actors, pp. 330–354. Membrane Computing: 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers, Springer International Publishing, Cham (2014), http://dx.doi.org/10.1007/978-3-319-14370-5_21

[11] Nicolescu, R.: Structured Grid Algorithms Modelled with Complex Objects, pp. 321–337. Membrane Computing: 16th International Conference, CMC 2015, Valencia, Spain, August 17-21, 2015, Revised Selected Papers, Springer International Publishing, Cham (2015), http://dx.doi.org/10.1007/978-3-319-28475-0_22

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

[13] Nishida, T.Y.: Membrane Algorithms, pp. 55–66. Membrane Computing: 6th International Workshop, WMC 2005, Vienna, Austria, July 18-21, 2005, Revised Selected and Invited Papers, Springer Berlin Heidelberg, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/11603047_4

[14] Păun, G.: Prerequisites, pp. 7–50. Membrane Computing: An Introduction, Springer Berlin Heidelberg, Berlin, Heidelberg (2002), https://doi.org/10.1007/978-3-642-56196-2_2

[15] Păun, G., Rozenberg, G.: An Introduction to and an Overview of Membrane Computing, chap. One, pp. 1–27. The Oxford Handbook of Membrane Computing, Oxford University Press, New York, NY, USA (2009)

[16] Smith, S.L., Imeson, F.: Glns: An effective large neighborhood search heuristic for the generalized traveling salesman problem. Computers & Operations Research 87, 1–19 (2017), http://www.sciencedirect.com/science/article/pii/S0305054817301223

[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 10(1), 89–99 (2015)

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

A Prolog simulation

Listing 1: Complete SWI-Prolog code for the cP Systems solution to the TSP

```
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]).
n(5).

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).
```

B F# simulation

Listing 2: Complete F# code for the cP Systems solution to the TSP

```
let infty = 10000

let E = // Specification of set of E objects
   [|  [| 0; 1; 3; infty; 2; |];
       [| 1; 0; infty; 6; 4; |];
       [| 3; infty; 0; 8; 5; |];
       [| infty; 6; 8; 0; 7; |];
       [| 2; 4; 5; 7; 0; |];
   |]

type s = int * Set<int> * list<int> * int // (r, u, p, c)
type z = list<int> * int // (p, c)

let visit (si:s) (u:Set<int>): list<s> =
    let r, _, f::p, c = si
    [for t in u do
        let w = E.[f].[t]
        if w < infty then
            yield (r, Set.remove t u, t :: f :: p, c + w) ]

let rec sgoal (slist: list<s>): list<s> =
    slist |> List.collect (fun si ->
        let r, u, _, _ = si
        if Set.isEmpty u
        then visit si (set [r])
        else sgoal (visit si u))

let hgoal (r:int) (y:Set<int>): list<z> =
    [(r, y, [r], 0)] |> sgoal |> List.map (fun (r, u, p, c) -> (p, c))

let minh (h: list<z>): z =
    h |> List.minBy (fun (p, c) -> c)

let go () =
    let r = 0
    let y = set [0..(Array.length E - 1)] |> Set.remove r
    hgoal r y |> minh |> printfn "%A"

go ()
```