Two Cycle-Related Problems of Regular Data Flow Graphs: Complexity and Heuristics

Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

Praveen K. Murthy
Edward A. Lee

(murthy,eal)@eecs.berkeley.edu

Technical Report: UCB/ERL M97/76, Electronics Research Lab

1 Abstract[1]

A regular data flow graph (RDFG) is a graph with a highly regular structure that enables its description to be exponentially smaller than the description size for an ordinary graph. Such graphs arise when certain regular iterative algorithms (like matrix multiplication or convolution) are modeled using dependence graphs. These graphs can be implemented either on systolic arrays, or wavefront arrays (WA). Systolic arrays have a global time clock; operations are scheduled statically and executed according to this schedule. The global clocking however, presents problems due to clock skewing in large circuits; hence, wavefront arrays are an attractive alternative. Wavefront arrays use a dataflow method of execution, and hence, do not require global synchronization. Array elements start computing whenever they have all of their inputs.

In a systolic implementation, the dependence graph cannot have any cycles since the existence of a schedule depends on the existence of a schedule vector that has non-negative dot product with each dependency edge. However, a graph implemented on a WA may have cycles provided that the cycles do not deadlock. There are a couple of computational problems that arise in this context: the first is the detection of deadlock; that is, to determine whether the graph to be implemented has a delay-free cycle. The second is to determine the maximum cycle mean; this represents the iteration rate with which the graph can be executed. While both of these problems are well known and well studied for ordinary static, homogeneous dataflow graphs, and can be solved with polynomial time algorithms, they have not been studied in the context of RDFGs. Since RDFGs have an exponentially more compact representation, we determine the complexity of these two problems in terms of this lower representation size. We show that the problems are

1. This research is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: The Alta Group of Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, NEC, Philips, and Rockwell.

NP-complete, and hence, no advantage can be theoretically gained from the smaller input size. We develop some heuristics that should work well even if not technically in polynomial time with respect to the specification size, especially for large RDFGs.

2 Introduction

A data flow graph (DFG) is a directed graph where the nodes represent computations, and the arcs communication channels and precedence constraints. Each node produces (consumes) one token onto (from) each of its output (input) arcs. An arc can have a number of **initial tokens** (also called **delays**). Each node has an associated **computation time**, that represents the number of cycles it takes to finish its computation. This dataflow model of computation is not the most general form of dataflow; more powerful versions include models where nodes can produce and consume a constant number of tokens of data per firing (i.e, not necessarily one), and models where nodes can produce and consume variable numbers of tokens per firing.

A path in the DFG is some connected sequence of edges in the graph. The beginning of the path is the node from which the first edge is taken, and the end of the path is the node that at which the last edge in the path ends. A cycle is a path whose beginning and end are the same. The total **delay count** of a path is the sum of delays on each of the edges in the path. The total computation time of the path is sum of the computation times of each of the nodes along the path. A DFG is said to be in a deadlocked state if there is at least one node in the graph that cannot be executed no matter how many times the other executable nodes are executed. A node cannot execute if it does not have at least one token on each of its input arcs. A DFG is said to be **strongly connected** if there is a directed path between any two nodes in the graph.

The following property about dataflow graphs is easily seen to be true:

Property 1: A DFG deadlocks if and only if there is some cycle whose delay count is 0.

Given a DFG, we are interested in the rate at which nodes can be executed. For acyclic DFGs, this rate is infinity since every execution of a node can proceed in parallel. If there are cycles in the graph, there is a well known lower bound on the achievable iteration period defined as

$$MAX_c\left\{\frac{T(c)}{D(c)}\right\},$$

where the maximum is taken over all cycles c in the graph, $T(c)$ is the computation time of cycle c and $D(c)$ its delay count. This quantity is known as the **maximum cycle mean (MCM)**. The throughput of the graph; that is, the rate at which nodes can be executed, is given by the inverse of the MCM. Even though the number of cycles in a DFG can be an exponential function of the size of the DFG, the MCM can be computed in polynomial time using a binary search and the Bellman-Ford shortest paths algorithm [9]. In other words, it is not actually necessary to enumerate cycles as the definition suggests.

3 Regular Data Flow Graphs

An RDFG [7] is a directed graph that can be characterized by embedding it in a finite dimensional index space such that each node of the graph resides at an index point. For an n-dimensional index space, we define the index vector as an n-tuple $i = \{i_1, \ldots, i_n\}$, where each i_k is an integer. A node can now be described by its location in the index space.

An RDFG has the following properties:

(1) It is defined over a contiguous, finite region of the index space.

(2) It has functionally identical nodes, with identical execution times at every index point in this region.

(3) For every node, each arc in the set of arcs for which the node is a terminal endpoint, has its initial point at the same relative offsets. This means that if there is an arc from a to b, then there is an arc from every point $a + x$ to every point $b + x$, where the addition for index points is the usual vector addition.

(4) For every node, the corresponding arcs have corresponding properties (namely, the number of initial tokens).

Figure 1 shows an example of an RDFG. The big dots on the arcs denote initial tokens. For “boundary” nodes, nodes that are near or at the edges of the region where the graph exists, not all incoming or outgoing arcs have a terminal node. In such a case, it is conceptually useful to think of these “hanging” arcs as input/output arcs from the graph; these are shown as gray arcs in the figure. An external device feeds in data

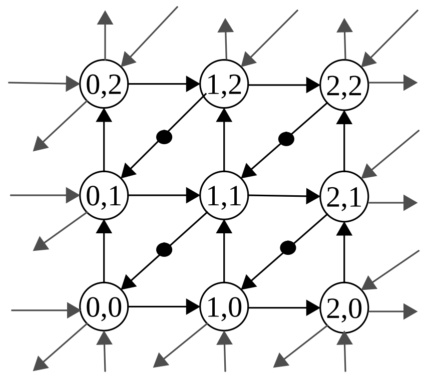


Fig 1. An example RDFG. The dots on the arcs represent initial tokens, or delays.

along the input arcs, and collects data from the output arcs. For the purposes of analysis, therefore, these boundary nodes with their input/output arcs will not matter, and will not be shown in the other graphs shown in this paper. Hence, the following property of an RDFG is obvious:

Property 2: Each node has the same number of incoming arcs as outgoing arcs.

An RDFG can be fully specified using an $n \times m$ *arc* matrix A for an n-dimensional index space where each node has m incoming arcs, and an m-dimensional vector D that specifies the number of initial tokens on each of the m arcs. In an n-dimensional index space, each arc is specified as an n-dimensional vector. Hence, the RDFG in Figure 1 is specified by

$$A = \begin{bmatrix} 1 & -1 & 0 \\ 0 & -1 & 1 \end{bmatrix}, \text{ and } D = [0, 1, 0].$$

In addition, we need to specify the region R of the space where the graph exists. For a rectangular region like below, it can be easily specified as two intervals whose cross product is the region where the graph resides. More complicated polygons can be specified by their vertices. In any case, it is easily seen that this description is very compact and highly scalable. In particular, if the number of arcs is constant, then the A and D matrices are constant even though the number of nodes may be arbitrarily big. In fact, as long as the number of arcs m is given by $m = O(\log N)$, where N is the number of nodes, an RDFG has an exponentially smaller description size than an ordinary data flow graph. For the example in Figure 1, the complete specification would be given as:

$$A, D, \text{ and } R = [0, N_1] \times [0, N_2].$$

The size of this description is clearly $O(\log m) + \log(N_1) + \log(N_2)$. If the graph were described using an adjacency list, the size would be $O(N_1 N_2 m)$ since there are $N_1 N_2$ nodes and each connects to m others (in this example $m = 3$). Clearly, $\log(m N_1 N_2)$ is exponentially smaller than $m N_1 N_2$.

A set of paths in the graph can be represented by non-negative integer vector $p^T = [p_1, \ldots, p_m]$, where p_i is the number of times an instance of arc a_i is traversed. If the path starts at a point x, then it ends at point $y = x + Ap$. For a cycle that ends at x, we have $y = x$ and hence, $Ap = 0$. So any vector in the null space of A is a potential candidate for a cycle. The sums of the node computation times in a path p can be computed as $p^T t$, where t is a vector whose entries are all the same; recall that all nodes have the same computation time in an RDFG. Since the computation times are the same, we can take it to be one without loss of generality. Similarly, the delay count of a path p is given by $p^T D$.

4 Cycle existence problems on RDFGs

In [7], Kung sets up the MCM problem as one of computing

$$MAX_c\left(1, \frac{c^T\bar{1}}{c^Td}\right) \quad \textbf{(EQ 1)}$$

where $\bar{1}$ is a vector of ones. This problem is stated to be an instance of nonlinear integer programming. Kung claims that since the size of the problem formulation is independent of the size of the graph, it can be solved more efficiently. However, Kung erroneously assumes in his comparison that the complexity of the MCM problem for DFGs is proportional to the number of cycles in the DFG. As already mentioned, this is untrue, and the MCM can be solved much more efficiently than enumerating cycles. Also, integer linear programming is an NP-complete problem, and integer non-linear programming is even worse. We can, of-course, represent an RDFG as a DFG and use a polynomial time algorithm (in the size of the DFG of-course) to compute the MCM. But we do not really know how this compares to the exponential complexity (in the size of the RDFG) of the non-linear integer programming formulation in equation 1.

Moreover, Kung apparently overlooks an important detail in his formulation given in equation 1: given a cycle vector c in the null space of arc matrix A, how do we know for sure that this cycle actually exists in the graph? Recall that the graph only exists in a finite region of the index space. Hence, it could be possible that there is no way to actually form the cycle suggested by c since the graph may not be big enough. Or, in the MCM computation, the c that maximizes the quantity in equation 1 may not exist in the graph at all. In the following, we show that the problem of determining whether a null vector c is a physical cycle is NP-complete. However, this does not mean that we have to resort to a DFG representation: as long as m is small enough, an exponential time algorithm in m might be better than a polynomial time one the DFG. However, the result establishes that there is theoretically little hope of making full use of the smaller description size of the RDFG.

To motivate the problem, consider the graph in Figure 2. The figure shows an RDFG specified as

$$A = \begin{bmatrix} -1 & 1 & 1 \\ 1 & 2 & -2 \end{bmatrix},\ D = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix},\ R = [0, 2] \times [0, 2].$$

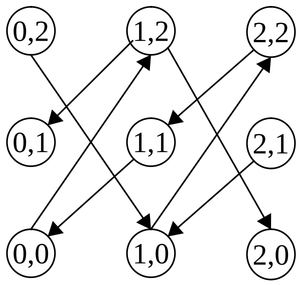


Fig 2. Example that shows the non-existence of a cycle in the region of the graph.

So the graph occupies the 3x3 square cornered at the origin. The vector $c^T = \begin{bmatrix}4 & 1 & 3\end{bmatrix}$ is in the null space of A, and hence is a potential cycle. However, this cycle does not actually exist in the graph because the graph is not big enough to enable traversing four instances of the arc $\begin{bmatrix}-1 & 1\end{bmatrix}^T$, and three instances of $\begin{bmatrix}1 & -2\end{bmatrix}^T$. So given a null space vector, we would like to efficiently determine whether the cycle physically exists in the region where the graph exists. For simplicity, we assume that the graph exists in a rectangular region. Formally,

Definition 1: The n-dimensional CYCLE EXISTENCE problem is the following. Given an $n \times m$ matrix A, and an integer vector c in the null space of A, and a rectangular region R given by $[0, n_1] \times \ldots \times [0, n_n]$, determine whether there is a point x in R and a path consisting of as many instances of each arc as specified by c so that the path never leaves the region R.

Definition 2: The PARTITION problem is the following. Given a set A of m positive integers, determine whether there is a subset S of A such that $\sum_{a \in S} a = \sum_{a \in A - S} a$.

Theorem 1: The PARTITION problem is NP-complete [6].

Lemma 1: In the 1-dimensional CYCLE EXISTENCE problem, a cycle starting from some point x in the region $R = [0, r]$ exists in R iff it exists starting from 0.

Proof: Clearly, if the cycle exists and goes through 0 (that is, the 0 node is in the cycle), then the cycle exists starting from 0. If the cycle does not go through 0, then the smallest index through which it goes is some number in R; let this be y. Clearly, we can subtract y from each index that is in the cycle and get one that goes through 0; in other words, we can translate any cycle back to the origin.

Note that the above lemma does not hold in higher dimensions because the numbers in each dimension cannot be treated independently.

The CYCLE EXISTENCE problem as defined may not even be in NP since given an arbitrary null vector, it is not clear whether the path (the precise order in which nodes are visited) can be written down succinctly. In other words, suppose that there are three arcs, a, b, c. Suppose the null vector happens to be $[2, 3, 3]^T$. However, the only way in which a cycle can be constructed in the region R might be via the sequence $abcbacbc$. It is not clear whether this string can be written down more compactly. In general, there might exist instances where the sequence of arcs traversed in the region has length given by the sum of the entries in the null vector; clearly, this is exponential in the size of the representation. However, we can still look at the complexity of this problem if we restrict our attention to null vectors whose entries are all bounded in some manner.

Definition 3: The 0-1-2 CYCLE EXISTENCE problem is the CYCLE EXISTENCE problem with the restriction that each entry in the null vector c has value drawn from the set $\{0, 1, 2\}$. That is, we only consider the set of cycles (i.e, a subset of the null space) where no arc occurs more than twice in a cycle.

Theorem 2: The 0-1-2 CYCLE EXISTENCE problem is NP-complete.

Proof: The problem is in NP since given a sequence of arcs in the path, and the point x, we have to just sum from x and ensure that we never go out of the region. This can be done in polynomial time since the total length of the path cannot be more than $2m$. To show completeness, we reduce from PARTITION. Let the m integers in an instance of PARTITION be $a_1, \ldots, a_m$. Define $b = (\sum a_i)/2$. Clearly, if a partition S exists, then each sum in

$$\sum_{a \in S} a = \sum_{a \in A-S} a$$

sums to b. Let the arc matrix in the instance of CYCLE EXISTENCE be defined as $A = \begin{bmatrix} a_1 & \ldots & a_m & -b \end{bmatrix}$. Hence, in this instance, $n = 1$. Clearly, $\begin{bmatrix} 1 & \ldots & 1 & 2 \end{bmatrix}^T$ is in the null space of A; hence, this is the null vector in our instance of 0-1-2 CYCLE EXISTENCE. Finally, let the region R be $[0, b]$. Thus, the RDFG in this specification has nodes at each integer point in $[0, b]$. The arc a_1 specifies that there is an arc between node 0 and a_1, between 1 and $1 + a_1$ and so on (see Figure 3). Clearly, a cycle that uses each arc a_i once, $-b$ twice, and still stays in R must have the property that there is a subset of the a_i that sum to b. Similarly, if such a subset exists, then we can find a cycle that never leaves R. **QED**.

Another problem is that of finding a “minimal” vector in the null space of A:

Definition 4: The MINIMAL CYCLE VECTOR is the following. Given an $n \times m$ matrix A, and an integer K, find a null space vector c such that the maximum element in c is less than K. Note that there is no requirement that the cycle represented by this vector actually exist; hence, no region is specified.

Corollary 1: The MINIMAL CYCLE VECTOR problem is NP-complete.

Proof: Membership in NP is trivial since it is just matrix-vector multiplication to verify that the result is the 0 vector. We use the same reduction from PARTITION; that is, the matrix A is as in the proof of Theorem

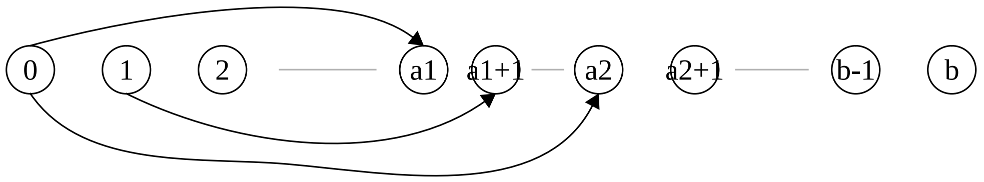


Fig 3. Graph used in the proof of Theorem 2.

2. We let $K = 2$. A cycle vector whose maximum entry is less than 2 for this A has to be a vector of 0-1 entries, and this would then solve the partition problem. Conversely, if the PARTITION problem has a solution, then such a vector would exist. **QED**.

In the discussion on membership of the general CYCLE EXISTENCE problem in NP, we mentioned that the presence of arbitrary integers in the null vector presents problems since there does not seem to be a way, in general, of representing such paths succinctly. In general, in combinatorial problems that have integers in their input instances, there are two sources of complexity: the number of discrete elements (the number of integers, the number of edges or nodes etc.), and the values the integers can take in the input instance. Hence, there are three “dimensions” to the complexity of the 1-d CYCLE EXISTENCE problem: the number of entries in the arc matrix m, the arbitrary values they take, and the potentially arbitrary values in c.

Suppose that m is fixed. Then, the PARTITION problem can be solved in polynomial time: we simply look at all possible subsets, and since m is not increasing, the number of subsets is constant. Hence, the arbitrariness of the a_i does not affect the complexity of PARTITION; only the number of them, m, does.

We know that with the entries in c restricted to be 0-1-2 valued, the CYCLE EXISTENCE problem is NP-complete, meaning that as m is increased, it is unlikely that an algorithm whose running time is a polynomial function of $\log(a_i)$, $\log(b)$ and m exists. If m is also fixed, then the 0-1-2 CYCLE EXISTENCE problem can be solved in polynomial time also; simply look at all possible ways of constructing the path. Since there are a fixed number of arcs, and each occurs at most two times, we can find out whether there is a sequence that never leaves the region. Hence, the values of the entries of A do not affect the complexity of 0-1-2 CYCLE EXISTENCE. So in reality, there are two “dimensions” to the complexity: m, and the arbitrariness of c. Hence, we look at the complexity of CYCLE EXISTENCE when m is fixed but c is allowed to have arbitrary entries.

Theorem 3: The 1-dimensional CYCLE EXISTENCE problem, with $m = 2$, is solvable in polynomial time.

Proof: Since $m = 2$, the arc matrix has two entries. One of the entries must be positive and one must be negative in order for there to be a non-negative null vector. Let the arc matrix $A = [a, -b]$. Let the region $R = [0, r]$. Any null vector c is of the form $c = \left[\frac{bk}{\overline{(a, b)}}\ \frac{ak}{\overline{(a, b)}}\right]^T$, where $\overline{(x, y)}$ denotes the gcd of x and y. Suppose that $k = 1$. Consider any path that traverses each of the two arcs as many times as given by c. In order for the path to stay in R, it can never visit a negative index, since the graph is only defined over $R = [0, r]$. From the set of paths that obey this constraint, we can choose the path where the maximum state visited is minimized. For example, let $A = [2, -3]$, and $c = \begin{bmatrix} 3 & 2 \end{bmatrix}^T$. One possible path that does not visit any negative state is $(2)(2)(-3)(2)(-3)$; all partial sums in this path are non-negative: $2, 2+2 = 4, 2+2-3 = 1, 2+2-3+2 = 3, 2+2-3+2-3 = 0$. Another possible path is $(2)(2)(2)(-3)(-3)$. The path $(2)(-3)(2)(2)(-3)$ visits a negative index (-1). Of the two possible paths that only visit non-negative indices, the maximum state visited by the first one is 4 while the maximum state visited by the second path is 6. Clearly, if we can compute the minimum maximum state visited over all such paths (where the minimum state visited is 0), we can answer yes or no to the existence question simply by comparing that value against r. It turns out that this problem is identical to the minimum buffer scheduling problem for a 2-node SDF graph [1]; there it is shown that the minimum maximum reached is given by $a + b - \overline{(a, b)}$. Hence, a cycle with cycle vector c exists iff $r \geq a + b - \overline{(a, b)}$. If $k > 1$, then we just repeat the path k times; this cannot increase the maximum state visited. Hence, the result holds for any cycle vector c. **QED**.

This result can be easily extended to 2 dimensions if the 2×2 arc matrix A has rank 1 (if it has rank 2, then the null space has dimension 0 and the 0 vector is the only cycle vector). In the 1-d case, if $m = 3$, the complexity of CYCLE EXISTENCE is open since the technique used above cannot be extended easily anymore. Hence, it does not seem possible to get a closed form expression for the minimum maximum state visited over all paths.

5 Heuristic approaches

Although we cannot apparently get polynomial-time algorithms for cycle detection/existence problems, we can adopt heuristic approaches that will definitely be much better in practice than resorting to a full-flown graph description. In particular, suppose that the graph is big enough, say 1000 by 1000 nodes. We investigate some ways of detecting deadlock efficiently (even if not in time polynomial in the size of the matrix descriptions), and computing the MCM.

Basically, we are interested in solving the following problem:

$$\text{find a non-negative, integer vector } x \text{ such that } Ax = 0 \qquad \textbf{(EQ 2)}$$

This will give us a vector that represents a cycle in a large enough graph. In the following, we study and review some fundamental properties of the above equation. We will use the notation $x = (x_1, \ldots, x_m)$ to denote an m-dimensional column vector.

5.1 Homogenous Linear Diophantine Equations

Equation 2 represents a system of linear equations whose coefficients are integers; these are called linear Diophantine equations. They are homogenous because the constant term is 0. Let

$$\Im = \{x : Ax = 0\}$$

be the set of solutions of $Ax = 0$. A solution x is called **irreducible** if it cannot be represented as a sum of other solutions in $\Im$. In the natural partial ordering of tuples ($x \le y \Leftrightarrow x_i \le y_i \forall i$), a solution y is irreducible iff there is no solution x such that $x \le y$. Hence, irreducible solutions are also called **minimal**; we will use the term minimal solution as it is easier to type. Clearly, $\Im$ is closed under addition and contains 0 as the zero element; hence, $\Im$ is an additive sub-monoid.

The following characterization is fundamental:

Theorem 4: The set $\Im$ has a **finite basis**; that is, there are a finite number of elements $h_i \in \Im$ such that any $x \in \Im$ can be represented as $\sum p_i h_i$, where p_i are all non-negative integers.

Proof: The theorem can be seen as a consequence of Hilbert's famous finite basis theorem; hence, the basis h_i is sometimes called the **Hilbert basis** for $\Im$ [13]. There is also a direct way of determining its finiteness; we outline a method due to Grace and Young [4]. Consider just one equation

$$a_1 x_1 + a_2 x_2 + \ldots + a_m x_m = b_1 y_1 + b_2 y_2 + \ldots + b_n y_n \qquad \textbf{(EQ 3)}$$

where the a_i, b_i are all positive integers, and we desire a non-negative solution (x, y), where $x = (x_1, \ldots, x_m)$ etc. Clearly, the following mn solutions are all minimal (the rest of the variables in each solution are 0):

$$x_r = b_s / g_{rs} \qquad y_s = a_r / g_{rs} \qquad g_{rs} = gcd(a_r, b_s) \qquad \forall (1 \le r \le m, 1 \le s \le n) \qquad \textbf{(EQ 4)}$$

We can bound the values of x_i in any minimal solution by

$$x_i \le b_1 + \ldots + b_n \qquad \textbf{(EQ 5)}$$

(the y case is symmetric). Indeed, suppose $x_i > b_1 + \ldots + b_n$ for some i. Then the right hand side of equation 3 is greater than $a_i(b_1 + \ldots + b_n)$, or $b_1(y_1 - a_i) + \ldots + b_n(y_n - a_i) > 0$, meaning that at least one of the $y_j > a_i$. Hence, this solution cannot be minimal since one of the solutions in equation 4 is smaller. Because of the bound, there are only a finite number of tuples that are minimal, and these will be the basis elements.

Suppose that there is another equation that also has to be satisfied. The minimal solutions of equation 3 can be written as

$$\begin{aligned} x &= \alpha_1 \qquad y = \beta_1 \\ &\dots \\ x &= \alpha_p \qquad y = \beta_p \end{aligned}$$

So any solution to equation 3 can be written as

$$\begin{aligned} x &= t_1\alpha_1 + \dots + t_p\alpha_p \\ y &= t_1\beta_1 + \dots + t_p\beta_p \end{aligned}$$

where t_i are arbitrary non-negative integers. We substitute the above values for x, y is the second equation, and get a new equation relating the t_i. This in turn has a finite basis, that can be determined by exhaustive search, to give the set of minimal solutions $t = \gamma_1, t = \gamma_2, \dots, t = \gamma_\sigma$. Any solution can be represented as $t = T_1\gamma_1 + \dots + T_\sigma\gamma_\sigma$, where the T_i are arbitrary non-negative integers. We substitute this back into the first equation to get

$$\begin{aligned} x &= \kappa_1 T_1 + \dots + \kappa_\sigma T_\sigma \\ y &= \lambda_1 T_1 + \dots + \lambda_\sigma T_\sigma \end{aligned}.$$

The minimal solutions are now just $x = \kappa_j, y = \lambda_j$. If there is a third equation, we can substitute the combination of these in and repeat the process. Since there are a finite number of equations in the system, the basis is finite, and can be determined in this fashion. **QED**

Example 1: Suppose $A = \begin{bmatrix} 1 & 1 & -1 & -1 \\ 2 & -2 & -1 & 1 \end{bmatrix}$. We need to determine the minimal solutions of $Ax = 0$, or

$$\begin{aligned} x_1 + x_2 &= x_3 + x_4 \\ 2x_1 + x_4 &= 2x_3 + x_3 \end{aligned}$$

The bound establishes that for the first equation, $x_i \le 2$ for each i in any minimal solution. By search, it is easily seen that $(1, 0, 1, 0), (1, 0, 0, 1), (0, 1, 1, 0), (0, 1, 0, 1)$ are the only minimal solutions. So any solution x can be written as $x = (t_1 + t_2, t_3 + t_4, t_1 + t_3, t_2 + t_4)$. Substituting this into the second equation yields $t_1 + 3t_2 = 3t_3 + t_4$. The minimal solutions to this equation are $(1, 0, 0, 1), (0, 1, 1, 0), (0, 1, 0, 3), (3, 0, 1, 0)$. So $t = (T_1 + 3T_4, T_2 + T_3, T_2 + T_4, T_1 + 3T_3)$. Substituting back, we get $x = (T_1 + T_2 + T_3 + 3T_4, T_1 + T_2 + 3T_3 + T_4, T_1 + T_2 + 4T_4, T_1 + T_2 + 4T_3)$. So the minimal solutions to the system are $(1, 1, 1, 1), (1, 1, 1, 1), (1, 3, 0, 4), (3, 1, 4, 0)$, obtained by setting each of the $T_i = 1, T_j = 0, j \ne i, i = 1, 2, 3, 4$. Eliminating redundant solutions gives the final set as $(1, 1, 1, 1), (1, 3, 0, 4), (3, 1, 4, 0)$.

The bound given by equation 5 (for one equation) was strengthened by Huet [5]:

$$x_i \le max(b_j), y_j \le max(a_i) \qquad \textbf{(EQ 6)}$$

Lambert [8] gives an even sharper bound:

$$x_1 + \ldots + x_m \le max(b_i), y_1 + \ldots + y_n \le max(a_i) \qquad \textbf{(EQ 7)}$$

It is easily seen that this bound is tight as there are minimal solutions that meet it with equality (for instance, the minimal solutions given for one equation in example 1).

Even with the sharpest bound, there can be an exponential number of minimal solutions even for one equation. The situation quickly gets worse in the procedure given in the proof above when multiple equations are involved. However, the number of minimal solutions to the whole system will be less than the number for any subset of the equations; so these bounds are ultimately not enough to tell us anything about the entire system.

Pottier [12] gives the following bound for systems of equations using geometric arguments. Let $\|M\|_\infty = max_{x \in \Im}\|x\|_\infty$, where $\|x\|_\infty = max(|x_i|)$. Let D_r be the largest absolute value of the minors of order r of A (a minor of order r of matrix A is the determinant of an $r \times r$ submatrix of A). Then,

$$\|M\|_\infty \le (n-r)D_r \qquad \textbf{(EQ 8)}$$

where n is the number of columns of A.

For the matrix in example 1, we get $D_1 = 2, D_2 = 4$. So $\|M\|_\infty \le 3 \times 2 = 6$ and $\|M\|_\infty \le 2 \times 4 = 8$. So 6 is the smallest bound in this case, and is not tight.

5.2 Deadlock detection

The graph deadlocks if there is a cycle c such that $c \in \Im$ and $c^T d = 0$ where d is the delay vector. Since both c, d are non-negative, $c^T d = 0$ iff $c_i = 0$ whenever $d_i > 0$ and $d_i = 0$ whenever $c_i > 0$. Setting $c_i = 0$ for an i where $d_i > 0$ eliminates column i from A; hence, the deadlock detection problem becomes one of solving a smaller system $A'c' = 0$, where A' is the submatrix A with the set of columns $\{a_i : d_i \ne 0\}$ removed. Deadlock occurs iff this system has a non-negative integral solution and the graph is big enough that the cycle exists in it. Techniques for determining a solution are given in 5.4.

5.3 Maximum cycle mean

The following lemma shows that minimal solutions are sufficient to determine the maximum cycle mean in the RDFG.

Lemma 2: Suppose that $c_1, c_2 \in \Im$, $\alpha, \beta > 0$, and $c = \alpha c_1 + \beta c_2$. Then

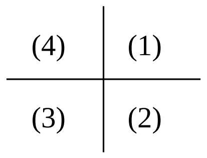
$$\frac{}{c^T d} \leq max\left\{\frac{}{c_1^T d}, \frac{}{c_2^T d}\right\}.$$

Proof: Recall that $\bar{1} = (1, 1, \ldots, 1)$. Letting $a_1 = c_1^T\bar{1}, a_2 = c_2^T\bar{1}, b_1 = c_1^T d, b_2 = c_2^T d$, let

$$\lambda_c = \frac{\alpha a_1 + \beta a_2}{\alpha b_1 + \beta b_2}, \lambda_1 = \frac{c_1^T\bar{1}}{c_1^T d} = \frac{a_1}{b_1}, \lambda_2 = \frac{c_2^T\bar{1}}{c_2^T d} = \frac{a_2}{b_2}.$$

We assume that $b_1, b_2 > 0$ since otherwise the cycle is deadlocked. Then $\alpha b_1\lambda_c + \beta b_2\lambda_c = \alpha a_1 + \beta a_2 = \alpha b_1\lambda_1 + \beta\lambda_2 b_2$. So $\alpha b_1(\lambda_c - \lambda_1) + \beta b_2(\lambda_c - \lambda_2) = 0$ implying that one of the terms is positive and one negative since $\alpha, \beta, b_1, b_2 > 0$. **QED.**

Hence, we do not need to consider non-minimal solutions since there will always be a minimal cycle that has larger MCM. One heuristic strategy for determining the MCM in a large enough graph is to use the bound on minimal solutions given in equation 8, and construct a “minimal” RDFG that provably contains all of the minimal cycles. On this graph, we can use standard graph techniques based on the Bellman-Ford shortest paths algorithm [9] for determining the MCM. However, constructing this minimal RDFG appears to be non-trivial; in the following we give one method that is not optimal in general. We then give a conjecture that, if correct, could give much more compact graphs.



Recall that we were restricting our attention to RDFGs specified over rectangular regions. One technique for constructing an RDFG that contains all of the minimal cycles is to simply map out a tour where each segment of the tour consists of $(n-r)D_r$ instances of each arc. The vectors are separated into four groups: group I contains the vectors that point in the Ith quadrant in the plane. The quadrants are shown in the figure to the right. The vectors are sorted by their gradients, with steepest first. We then construct a walk by taking all of the vectors in group (1), in the sorted order. We continue the walk by choosing vectors in group (2) in reverse sorted order, then group (3) vectors in sorted order, and finally group (4) vectors in reverse sorted order. We call this the maximal tour. The RDFG is then created over the smallest rectangle that contains this tour. This procedure can be easily generalized to higher dimensional RDFGs.

Suppose now that c is some minimal cycle. Clearly, the maximum horizontal distance that this cycle covers has to less than the horizontal distance spanned by the maximal tour given above. Similarly, the maximum vertical distance spanned by any minimal cycle is also less than that spanned by the maximal tour. Hence, the minimal cycle is contained in a rectangle that is smaller than the rectangular region over which the graph exists in the sense that the rectangle containing the minimal cycle is contained in the graph region. Hence, the constructed graph region contains all of the minimal cycles.

The bound given in equation 8 is not tight, as mentioned, and can thus lead to graphs much bigger than necessary. Even if the bound were tight, the graph constructed might still be bigger than necessary since we do not need to traverse each type of edge in series as is done in the construction above. The following example elucidates this.

Example 2: Suppose $A = \begin{bmatrix} 1 & 1 & -1 & -1 \\ 2 & -2 & -1 & 1 \end{bmatrix}$. The minimal solutions that we determined in example 1 were: $(1, 1, 1, 1), (1, 3, 0, 4), (3, 1, 4, 0)$. The computed bound is $\|M\|_\infty \le 3 \times 2 = 6$, whereas the tight bound, by examining the minimal solutions, is 4. Using the tight bound of 4, we construct the graph by constructing the tour, starting from an arbitrary point, of four instances of $\begin{bmatrix} 1 & 2 \end{bmatrix}^T$, then four of $\begin{bmatrix} 1 & -2 \end{bmatrix}^T$ and so on. The maximum horizontal distance spanned is 8 and the maximum vertical distance is 12. Hence, the "minimal" graph constructed in this case exists over an 8×12 rectangle and contains 96 nodes.

However, consider the 3×4 graph shown in figure 4. This graph contains both of the cycles $(1, 3, 0, 4), (3, 1, 4, 0)$:
$(0, 1) \to (1, 3) \to (2, 1) \to (1, 2) \to (0, 3) \to (1, 1) \to (0, 2) \to (1, 0) \to (0, 1)$, and
$(0, 1) \to (1, 3) \to (2, 1) \to (1, 0) \to (2, 2) \to (1, 1) \to (0, 0) \to (1, 2) \to (0, 1)$. Of course, it contains $(1, 1, 1, 1)$: $(0, 0) \to (1, 2) \to (2, 0) \to (1, 1) \to (0, 0)$. So, this 3×4 graph is the smallest graph that contains all of the minimal cycles for this example, and the 8×12 graph constructed above is larger than necessary.

One reason why the graph that results from our construction is so big is our insistence on spanning the maximum possible distance horizontally and vertically. This is done to make the proof of correctness simple, but we conjecture that the following is also true:

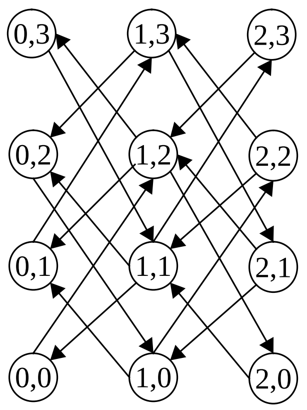


Fig 4. Minimal graph for example 2.

Conjecture 1: A minimal cycle c exists in the graph if it is possible to find a path in the graph that traverses each arc $\|M\|_\infty$ times, without repeating any arc.

If the conjecture is true, then we can construct a "minimal" tour where we attempt to stay as close to the starting point as possible, without repeating any arcs. The smallest rectangle that contains the resulting tour is then the "minimal" graph. For example 2, the graph constructed using this type of heuristic (with the tight bound of 4) results in an 3×5 graph, which is quite close to the optimal solution.

Note that the final qualifier in the conjecture above, "without repeating any arc", is necessary as otherwise, the minimal tour for the above example would result in an 3×2 graph that does not contain all of the minimal cycles.

Another heuristic is to simply enumerate the cycles. This is done by enumerating the minimal solutions in $\Im$; algorithms are given in [2][3][12]. It is not clear which approach will be more beneficial in practice.

5.4 Integer programming techniques

Since equation 2 is an instance of integer linear programming (ILP), we review some techniques that can also be used to solve it; in particular, these techniques are needed to solve the deadlock problem as defined in section 5.2.

A matrix U is called **unimodular** if it is integral and has a determinant of ± 1. A matrix of full row rank is said to be in **Hermite normal form (HNF)** if it has the form $\begin{bmatrix} B & 0 \end{bmatrix}$, where B is a non-singular, lower triangular matrix, in which $b_{ii} > 0 \ \ \forall i$, $b_{ij} \le 0$ and $|b_{ij}| < |b_{ii}|$ for $i > j$.

Theorem 5: If A is an $m \times n$ integer matrix, with $rank(A) = m$, then there exists an $n \times n$ unimodular matrix C such that $AC = \begin{bmatrix} B & 0 \end{bmatrix}$ where B is in HNF, and $H^{-1}A$ is an integer matrix.

Proof: See [11] for a proof.

The HNF can be determined in polynomial time.

The basic integer linear programming optimization problem is the following:

$$max\{cx : Ax \le b\} \qquad \textbf{(EQ 9)}$$

where A, b, c are integral matrices and vectors, the solution vector x is required to be integral. The associated feasibility problem is to find an integer vector x such that the following holds, where again the matrices and vectors are all integral:

$$\{x : Ax \le b\} \qquad \textbf{(EQ 10)}$$

It can be easily shown that if the feasibility problem can be solved, than the optimization problem can be solved using a polynomial number of calls of the feasibility solving algorithm. The feasibility problem is, however, NP-complete. So is the following feasibility problem:

$$\{x : Ax = b, x \ge 0\} \qquad \textbf{(EQ 11)}$$

where we desire a non-negative vector x.

5.5 Determining the null vector efficiently using Lenstra's algorithm

Interestingly enough, if the number of columns of the matrix A is fixed, then all of the above problems can be solved in polynomial time by Lenstra's algorithm [10]. In other words, the complexity of these problems does not arise because of the arbitrary integers that are allowed in the problem instance; rather, they arise due to the combinatorial aspect: the number of columns. The precise statement of Lenstra's theorem is:

Theorem 6: [Lenstra] For each fixed natural number n, there exists a polynomial time algorithm which solves the ILP problem in equation 9, where A has rank at most n, and where input data are all rational.

Hence, if either the number of columns or the number of rows is fixed, then the ILP problem in equation 9 can be solved in polynomial time. From the above, we can prove the following:

Corollary 2: For fixed natural numbers n and m, there exists a polynomial time algorithm which solves the ILP problem in equation 11, where A is $n \times m$, and where input data are all rational.

Proof: Suppose that A has full row rank n, and has m columns. Then we can find an $m \times m$ unimodular matrix U such that $AU = \begin{bmatrix} B & 0 \end{bmatrix}$. For any integer vector x, there is an integer vector y such that $x = Uy$. Hence, $Ax = b$ iff $AUy = b = \begin{bmatrix} B & 0 \end{bmatrix} y$. Let $U = \begin{bmatrix} U_1 & U_2 \end{bmatrix}$, where U_1 is $m \times n$, and U_2 is $m \times (m-n)$, and $y = \begin{bmatrix} y_1 & y_2 \end{bmatrix}^T$ where y_1 has n rows and y_2 has $m-n$ rows. We can solve for $y_1 = B^{-1}b$. We need to determine y_2 such that $Uy = x \ge 0$, or $-U_2 y_2 \le U_1 y_1$. This last problem is an ILP problem in $m-n$ variables, and can be solved using Lenstra's algorithm. **QED**.

Practically, the conditions of the corollary will usually hold since the number of rows in the arc matrix is some small fixed number (like 2 or 3 for 2 or 3 dimensional RDFGs), and the number of columns is also likely to be small in practice (since each node will have small degree). Hence, we can find a non-negative null vector, representing a potential cycle, efficiently in such cases.

6 Conclusion

We have studied two problems concerning cycles in RDFGs in this paper: deadlock detection and computing the maximum cycle mean. Since RDFGs can be represented very compactly, our intention was to study the complexity of these two problems in terms of the compact representation. We have shown that these problems are NP-complete. We have then given some heuristic techniques for determining these quantities; these techniques should be better than those currently known in the literature, although we have made no attempt to quantify the improvement, if any, in this paper.

7 References

[1] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.

[2] E. Contejean, H. Devie, “An Efficient Incremental Algorithm for Solving Systems of Linear Diophantine Equations,” Information and Computation, Vol. 113, No. 1, August 1994.

[3] E. Domenjoud, “Solving Systems of Linear Diophantine Equations: An Algebraic Approach,” Mathematical Foundations of Computer Science, 16th Intl. Symposium, Kazinierz, Dolny, Poland, September 1991.

[4] J. H. Grace, A. Young, *The Algebra of Invariants*, Cambridge University Press, 1903.

[5] G. Huet, “An Algorithm to Generate the Basis of Solutions to Homogenous Linear Diophantine Equations,” Information Processing Letters, Vol. 7, No. 3, April, 1978.

[6] R. M. Karp, “Reducibility Among Combinatorial Problems,” Complexity of Computer Computations, Miller and Thatcher Eds, Plenum Press, NY, 1972

[7] S. Y. Kung, *VLSI Array Processors*, Prentice Hall, 1988.

[8] J. L. Lambert, “Un Probleme d’accessibilite dans les reseaux de Petri,” Ph.D thesis, University of Paris-Sud, Orsay, France, 1987.

[9] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, 1976.

[10] H. W. Lenstra, “Integer Programming with a Fixed Number of Variables,” Mathematics of Operations Research, Vol. 8, 1983.

[11] G. Nemhauser, L. Wolsey, *Integer and Combinatorial Optimization*, Wiley, 1988.

[12] L. Pottier, “Minimal Solutions of Linear Diophantine Systems: Bounds and Algorithms”, Rewriting Techniques and Applications, 4th International Conference, RTA-91 Proceedings, Como, Italy, 10-12 April 1991.

[13] A. Schrijver, *Theory of Linear and Integer Programming*, Wiley, 1986.