A Heuristic for Direct Product Graph Decomposition

Luca Calderoni[1] ⓘ *Luciano Margara*[1] ⓘ *Moreno Marzolla*[1,2] ⓘ

[1]Dept. of Computer Science and Engineering,
University of Bologna, Italy
[2]Center for Inter-Department Industrial Research ICT, Università di Bologna, Italy

Abstract. In this paper we describe a heuristic for decomposing a directed graph into factors according to the direct product (also known as *Kronecker*, *cardinal* or *tensor* product). Given a directed, unweighted graph G with adjacency matrix $\mathbf{Adj}(G)$, our heuristic aims at identifying two graphs G_1 and G_2 such that $G = G_1 \times G_2$, where $G_1 \times G_2$ is the direct product of G_1 and G_2. For undirected, connected graphs it has been shown that graph decomposition is "at least as difficult" as graph isomorphism; therefore, polynomial-time algorithms for decomposing a general directed graph into factors are unlikely to exist. Although graph factorization is a problem that has been extensively investigated, the heuristic proposed in this paper represents – to the best of our knowledge – the first computational approach for general directed, unweighted graphs. We have implemented our algorithm using the MATLAB environment; we report on a set of experiments that show that the proposed heuristic solves reasonably-sized instances in a few seconds on general-purpose hardware. Although the proposed heuristic is not guaranteed to find a factorization, even if one exists; however, it always succeeds on all the randomly-generated instances used in the experimental evaluation.

1 Introduction

Decomposition of complex structures into simpler ones is one of the driving principles of mathematics and applied sciences. Everybody is familiar with the idea of integer factorization, a topic that is actively studied due to its number-theoretic as well as practical implications, e.g., in cryptography. The concept of factorization can be applied to other mathematical objects as well, such as graphs.

E-mail addresses: luca.calderoni@unibo.it (Luca Calderoni) luciano.margara@unibo.it (Luciano Margara) moreno.marzolla@unibo.it (Moreno Marzolla)

Once the concept of "graph product" is defined, one may naturally ask whether a graph G can be decomposed into the product of two (or more) smaller graphs.

Graph products are an active area of research because they are involved in a number of computer science applications, such as load balancing in distributed systems [2], network analysis [11], symbolic computation [4], and quantum computing [9]. The most common types of graph products that have been investigated in the literature are: *Cartesian product*, *Direct product*, *Strong product* and *Lexicographic product*. Of these, the *Direct product*, also known as *Kronecker* or *Cardinal product*, is widely used and will be the focus of this paper. For example, the Kronecker product of graphs has applications related to graph coloring, decomposition, embedding and matching [1, 8], automata theory [5], concurrency modeling in multiprocessor systems [10] and computer networks [11]. We use the symbol $\times$ to denote the direct product, e.g., $G = G_1 \times G_2$.

It has recently been shown in [3] that deciding whether an undirected, unweighted, non-bipartite graph G is composite according to the direct product, i.e., whether there exist nontrivial graphs G_1, G_2 such that $G = G_1 \times G_2$, is at least "as difficult as" deciding whether two graphs are isomorphic (a graph is *nontrivial* if it has more than one node). More formally, the graph isomorphism problem is polynomial-time many-one reducible to the graph compositeness testing problem (the complement of the graph primality testing problem). A consequence of this result is that the graph isomorphism problem for undirected, non-bipartite graphs is polynomial-time Turing reducible to the primality testing problem. It is therefore unlikely that there exists a polynomial-time algorithm for graph factorization according to the direct product, unless graph isomorphism is in P, i.e., graph factorization is graph isomorphism-hard.

The problem of graph factorization has been extensively studied [6] from the theoretical point of view: mathematical properties of graph products are known, as well as factorization algorithms for a few special cases. For example, it is known that prime factorization of an undirected, connected and non-bipartite graph with n nodes and m edges can be found in time $O(mn^2)$ [7]. However, in [3] it is shown that prime factorization of an undirected, unconnected, non-bipartite graph is graph isomorphism-hard, suggesting that the lack of connectedness plays a major role in making direct product primality testing and factorization harder.

Despite the large volume of theoretical work, the problem of graph factorization has not received yet much attention from the experimental research community. Indeed, to the best of our knowledge, no implementation of graph factorization algorithms for general directed graphs is available. The problem is exacerbated by the fact that the existing algorithms only work on special kinds of graphs, and it is not known whether they can be generalized to arbitrary directed graphs, or whether different algorithms exist for general graphs. In this paper we begin to bridge the gap between theory and practice by proposing a heuristic for direct-product factorization of general graphs: given a directed, unweighted graph G with n nodes and two positive integers n_1, n_2 such that $n = n_1 \times n_2$, our algorithm looks for two nontrivial graphs G_1, G_2 with n_1 and n_2 nodes, respectively, such that $G = G_1 \times G_2$, provided that such graphs exist. Our heuristic is based on gradient-descent local search. As such, it is not guaranteed that it finds a solution, even if one exists; however, we carried out a set of computational experiments that show that our algorithm does find a valid solution quickly in many cases.

To the best of our knowledge, the heuristic described in this paper is the first computational solution to the problem of factorization of general unweighted graphs, i.e., graphs whose structure is not subject to any constraint.

2   Notation and Basic Definitions

A directed graph $G = (V, E)$ is described as a finite set V of nodes $V = \{v_1, \ldots, v_n\}$ and a finite set of edges $E \subseteq V \times V$, where an edge $e \in E$ is an ordered pair $e = (u, v)$, $u, v \in V$; an edge of the form (v, v) is called *self loop* or simply *loop*. Given a graph G, $V(G)$ and $E(G)$ are the set of nodes and edges of G, respectively. We denote by $G_1 \cup G_2$ the disjoint union of graphs G_1 and G_2, i.e., the graph with node set $V(G_1) \cup V(G_2)$ and edge set $E(G_1) \cup E(G_2)$; disjoint means that $V(G_1) \cap V(G_2) = \emptyset$.

Four types of graph products have been investigated in the literature: *Cartesian product*, *Direct product*, *Strong product* and *Lexicographic product*. In all cases, the product of two graphs G_1, G_2 is a new graph G whose set of nodes is the Cartesian product of $V(G_1)$ and $V(G_2)$:

$$V(G) = V(G_1) \times V(G_2) = \{(u, v) \mid u \in V(G_1) \wedge v \in V(G_2)\}$$

For the *Cartesian product* $G_1 \square G_2$, the edge set is defined as

$$E(G_1 \square G_2) = \{\{\{x, y\}, \{x', y'\}\} \mid (x = x' \wedge \{y, y'\} \in E(G_2)) \vee (\{x, x'\} \in E(G_1) \wedge y = y')\}$$

For the *Strong product* $G_1 \boxtimes G_2$, the edge set is defined as

$$E(G_1 \boxtimes G_2) = E(G_1 \square G_2) \cup E(G_1 \times G_2)$$

For the *Lexicographic product* $G_1 \circ G_2$, the edge set is defined as

$$E(G_1 \circ G_2) = \{\{\{x, y\}, \{x', y'\}\} \mid \{x, x'\} \in E(G_1) \vee (x = x' \wedge \{y, y'\} \in E(G_2))\}$$

In this paper we are concerned with the *Direct product*, also known as Kronecker or cardinal product. The direct product of two graphs G_1, G_2 is denoted as $G = G_1 \times G_2$, where the edge set is defined as

$$E(G_1 \times G_2) = \{((x, y), (x', y')) \mid (x, x') \in E(G_1) \wedge (y, y') \in E(G_2)\}$$

Figure 1 shows an example of the direct product of two graphs G_1, G_2.

The edge set of an unweighted graph G can be represented as an adjacency matrix $\mathbf{Adj}(G)$. If G has n nodes, its adjacency matrix $\mathbf{M} = \mathbf{Adj}(G)$ is an $n \times n$ binary matrix, where $m_{ij} = 1$ if and only if $(v_i, v_j) \in E$. We denote the set of binary matrices of size $n \times m$ as $\mathbb{B}^{n \times m}$, $\mathbb{B} = \{0, 1\}$.

The algorithm described in the paper relies on the fact that the direct product $G_1 \times G_2$ of two graphs can be expressed in terms of the *Kronecker product* of their adjacency matrices [6, 14]. Given an $n \times m$ matrix $\mathbf{B}$ and a $p \times q$ matrix $\mathbf{C}$, the Kronecker product $\mathbf{A} = \mathbf{B} \otimes \mathbf{C}$ is an $np \times mq$ matrix that is composed of $n \times m$ blocks of size $p \times q$, each block being the product of elements of $\mathbf{B}$ and the whole matrix $\mathbf{C}$:

$$\mathbf{A} = \mathbf{B} \otimes \mathbf{C} = \begin{pmatrix} b_{11}\mathbf{C} & b_{12}\mathbf{C} & \ldots & b_{1m}\mathbf{C} \\ b_{21}\mathbf{C} & b_{22}\mathbf{C} & \ldots & b_{2m}\mathbf{C} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1}\mathbf{C} & b_{n2}\mathbf{C} & \ldots & b_{nm}\mathbf{C} \end{pmatrix} \tag{1}$$

Note that if $\mathbf{B}, \mathbf{C}$ are binary matrices, then $\mathbf{A} = \mathbf{B} \otimes \mathbf{C}$ will be as well. The relation between the direct product of graphs and the Kronecker product of their adjacency matrices is expressed by the following Lemma.
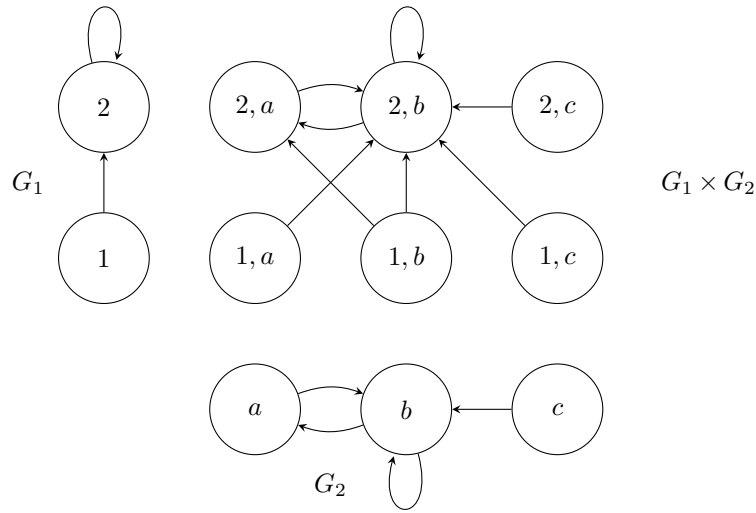
Figure 1: Direct product $G = G_1 \times G_2$

Lemma 2.1 *Given two directed, unweighted graphs G_1 and G_2, then*

$$\mathbf{Adj}(G_1 \times G_2) = \mathbf{P}^\mathsf{T}\left(\mathbf{Adj}(G_1) \otimes \mathbf{Adj}(G_2)\right)\mathbf{P}$$

where $\mathbf{P}$ is a suitable permutation matrix.

We recall that a *permutation matrix* $\mathbf{P} \in \mathbb{B}^{n \times n}$ is a square binary matrix with exactly a single 1 on each row and column. In other words, the adjacency matrix of the direct product of G_1, G_2 is equal to the Kronecker product of the adjacency matrices of G_1 and G_2, up to a rearrangement (relabeling) of the nodes of the resulting graph $G_1 \times G_2$.

For example, for the graphs in Figure 1 we have:

$$\mathbf{Adj}(G_1) = \begin{array}{c} \\ 1 \\ 2 \end{array}\begin{array}{c} 1 \quad 2 \\ \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \end{array} \qquad\qquad \mathbf{Adj}(G_2) = \begin{array}{c} \\ a \\ b \\ c \end{array}\begin{array}{c} a \quad b \quad c \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

$$\mathbf{Adj}(G_1 \times G_2) = \begin{array}{c} \\ 1,a \\ 1,b \\ 1,c \\ 2,a \\ 2,b \\ 2,c \end{array}\begin{array}{c} 1,a \quad 1,b \quad 1,c \quad 2,a \quad 2,b \quad 2,c \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Symbol	Meaning
$\mathbb{B}$	The set $\{0, 1\}$
$\mathbf{Adj}(G)$	The adjacency matrix of graph G
$\mathbf{P}$	Permutation matrix
$\times$	The direct graph product operator
$\otimes$	The Kronecker matrix product operator
$\phi(\mathbf{A}\,\mathbf{B}, \mathbf{C})$	Metric that shows "how well" $\mathbf{A}$ can be written as the Kronecker product $\mathbf{B} \otimes \mathbf{C}$ (lower is better)

Table 1: Summary of notation.

In this case the permutation matrix is the identity matrix, since we are assuming the mapping $(1, a) \rightarrow 1, (1, b) \rightarrow 2, (1, c) \rightarrow 3, \ldots$ of nodes into indices. A different mapping would require relabeling the nodes of $G_1 \times G_2$.

A *nontrivial* graph G is a graph with more than one node ($|V(G)| > 1$). We say that a graph G is *prime* according to a given graph product $\odot$ if G is nontrivial and $G = G_1 \odot G_2$ implies that either G_1 or G_2 are trivial, i.e., one of them has exactly one node.

Finally, we introduce the concept of *block matrix* that will be used in the following to discuss several subroutines of our heuristic.

Definition 2.2 *Let us consider a binary matrix $\mathbf{A}$ made of $dimB \times dimB$ submatrices (or blocks) $\mathbf{C}_{ij}$ of size $dimC \times dimC$ each. Let μ be the average number of ones over all blocks[1], and let s_{ij} be the number of ones in $\mathbf{C}_{ij}$; we say that $\mathbf{A}$ is a* block matrix *if and only if for each block $\mathbf{C}_{ij}$:*

$$s_{ij} = \begin{cases} x, & \text{if } x > \mu \\ 0, & \text{otherwise} \end{cases}$$

In other words, a *block matrix* is made of blocks such that each block is either entirely zero, or has a number of 1s that is strictly above the average number of 1s over all blocks. Fore ease of clarity, some examples of block matrices are provided in Figure 3f and in Figure 4c.

Table 1 summarizes the notation used in this paper.

3 A Heuristic for Direct Product Factorization

In this section we describe a heuristic for searching for a nontrivial decomposition $G = G_1 \times G_2$ of a directed, unweighted graph G. The heuristic is not guaranteed to find such a decomposition, even when one exists. We assume that the size (number of nodes) of G_1 and G_2 is known; if this is not the case, then by the definition of direct product it must hold that $n = n_1 \times n_2$ where n, n_1, n_2 are the number of nodes of G, G_1, G_2 respectively; therefore, both n_1 and n_2 must be nontrivial divisors of n. Since the number of divisors of n is bounded from above by n, we can brute-force all combinations of n_1, n_2, whose number grows polynomially w.r.t. n. Therefore, the algorithm performance would be impacted by (at most) a polynomial factor in n.

The basic idea is to take advantage of Lemma 2.1 to find a suitable permutation matrix $\mathbf{P}$ so that the (permuted) adjacency matrix of G is in block form and can therefore be written as the

---

[1] The average number of ones over all blocks is computed considering, for each block, the number of elements set to one. Each of these values is then summed together and finally divided by the number of blocks.

Kronecker product of two smaller matrices $\mathbf{B}$ and $\mathbf{C}$; these matrices can then be interpreted as the adjacency matrices of two graphs G_1, G_2 with the property that $G = G_1 \times G_2$. The problem is equivalent to finding a graph G' isomorphic to G such that $\mathbf{Adj}(G') = \mathbf{B} \otimes \mathbf{C}$; this is not surprising, given the relationship between graph isomorphism and compositedness testing proven in [3].

One more ingredient is needed to complete the heuristic, i.e., a way to decide whether a binary square matrix can be written as the Kronecker product of two smaller matrices. This is an instance of the more general *nearest Kronecker product* (NKP) problem [13, 12]: given $\mathbf{A} \in \mathbb{R}^{m \times n}$ with $m = m_1 m_2$, $n = n_1 n_2$, find $\mathbf{B} \in \mathbb{R}^{m_1 \times n_1}$ and $\mathbf{C} \in \mathbb{R}^{m_2 \times n_2}$ such that

$$\phi(\mathbf{A}, \mathbf{B}, \mathbf{C}) = \|\mathbf{A} - \mathbf{B} \otimes \mathbf{C}\| \tag{2}$$

is minimized, according to some norm $\| \cdot \|$. If $\mathbf{A}$ is the Kronecker product of $\mathbf{B}$ and $\mathbf{C}$, then the minimum norm would be zero ($\phi(\mathbf{A}, \mathbf{B}, \mathbf{C}) = 0$). The NKP problem (2) can be solved by computing a Singular Value Decomposition (SVD) of a suitably reshaped and permuted version of matrix $\mathbf{A}$ (see [13] for details).

To recap: to decompose a directed, unweighted graph G with adjacency matrix $\mathbf{M} = \mathbf{Adj}(G)$, we need to find a permutation of $\mathbf{M}$, say $\mathbf{P}^\mathsf{T}\mathbf{M}\mathbf{P}$, and two square matrices $\mathbf{M}_1 = \mathbf{Adj}(G_1)$ and $\mathbf{M}_2 = \mathbf{Adj}(G_2)$ of given sizes such that:

$$0 = \phi(\mathbf{P}^\mathsf{T}\mathbf{M}\mathbf{P}, \mathbf{M}_1, \mathbf{M}_2)$$
$$= \|\mathbf{P}^\mathsf{T}\mathbf{M}\mathbf{P} - \mathbf{M}_1 \otimes \mathbf{M}_2\|$$

Algorithm 1: Direct product decomposition

Input: $\mathbf{A} \in \mathbb{B}^{n \times n}$, $n = n_1 \times n_2$, K, *tol*
Output: $\mathbf{P} \in \mathbb{B}^{n \times n}$, $\mathbf{B} \in \mathbb{B}^{n_1 \times n_1}$, $\mathbf{C} \in \mathbb{B}^{n_2 \times n_2}$ or "no solution found"
$\mathbf{P}$ = Identity matrix of size $n \times n$;
repeat
    for $i = 1, \ldots, K$ **do**
        Let $\mathbf{P}_i$ be obtained from $\mathbf{P}$ by exchanging a random pair of rows/columns;
        Find $\mathbf{B}_i \in \mathbb{B}^{n_1 \times n_1}, \mathbf{C}_i \in \mathbb{B}^{n_2 \times n_2}$ s.t. $r_i = \phi(\mathbf{P}_i^\mathsf{T}\mathbf{A}\mathbf{P}_i, \mathbf{B}_i, \mathbf{C}_i)$ is minimized;
    end
    Let $j = \arg\min_i\{r_i\}$;
    $\mathbf{P} = \mathbf{P}_j$;
until $(r_j < tol) \vee$ (*number of iterations* $\geq$ *threshold*);
if $r_j < tol$ **then**
    return $\mathbf{P}, \mathbf{B}_j, \mathbf{C}_j$;
else
    return "no solution found";
end

With a slight abuse of notation, in the following we write $\phi(\mathbf{A})$ to denote the minimum value of $\phi(\mathbf{A}, \mathbf{B}, \mathbf{C})$ that can be achieved by suitably choosing $\mathbf{B}, \mathbf{C}$; in other words, $\phi(\mathbf{A})$ shows how well matrix $\mathbf{A}$ can be expressed as the Kronecker product of smaller matrices $\mathbf{B}, \mathbf{C}$ of given sizes. $\phi()$ is a *lower-is-better* metric: if $\phi(\mathbf{A}) = 0$, the matrix $\mathbf{A}$ is in Kronecker form.

Note that we must impose the additional constraint that $\mathbf{M}_1$ and $\mathbf{M}_2$ must be binary matrices, which is not guaranteed by the algorithm for Kronecker factorization described in [13]. For this reason we developed a novel algorithm for solving the NKP problem on binary matrices, described below.

Algorithm 1 shows a very high-level overview of the proposed heuristic. The algorithm uses the gradient-descent technique to find the permutation $\mathbf{P}$ such that the matrix $\mathbf{P}^\mathsf{T}\mathbf{AP}$ can be decomposed according to the Kronecker product.

Matrix $\mathbf{P}$ is built incrementally, starting from the identity permutation, by exchanges of rows and columns. At each step, the algorithm tries to decrease the value of $\phi(\mathbf{P}^\mathsf{T}\mathbf{AP})$ so that either the value becomes zero (in which case we have found a decomposition of the input graph), or no optimal permutation is found within the allotted number of steps.

To limit the search space, at each step the algorithm generates K random permutations of the current matrix $\mathbf{P}$; for each permutation $\mathbf{P}_i$, the algorithm solves the NKP problem by identifying two binary matrices $\mathbf{B}_i, \mathbf{C}_i$ such that

$$r_i = \phi(\mathbf{P}_i^\mathsf{T}\mathbf{AP}_i, \mathbf{B}_i, \mathbf{C}_i) = \|\mathbf{P}_i^\mathsf{T}\mathbf{AP} - \mathbf{B}_i \otimes \mathbf{C}_i\|$$

is minimized. If the minimum value r_j is not zero (within a user-defined tolerance tol), the process is repeated starting from the "best" permutation matrix $\mathbf{P}_j$ found so far. If no solution is found after some maximum number of iterations, the procedure assumes that matrix $\mathbf{A}$ can not be expressed as the Kronecker product of two matrices of size $n_1 \times n_1$ and $n_2 \times n_2$.

Despite its appealing simplicity, a direct implementation of Algorithm 1 is not effective in solving the graph decomposition problem. First of all, gradient-descent procedures may become stuck in a local minimum, with the result that they fail to find a global optimum (in our case, the global optimum is a permutation $\mathbf{P}_j$ for which $r_j = 0$, provided that such a permutation exists). Another issue, already stated above, is that we need to solve a modified version of the NKP problem, in which the factors $\mathbf{B}$ and $\mathbf{C}$ must be binary matrices.

Algorithm Details   We now provide the detailed description of our graph decomposition heuristic, where both these issues will be addressed. The complete pseudocode of the heuristic is provided in the following, while a MATLAB implementation along with the source code can be downloaded from https://github.com/calderonil/kron.

As already shown in (1), the Kronecker product $\mathbf{A} = \mathbf{B} \otimes \mathbf{C}$ of two binary matrices $\mathbf{B}$ and $\mathbf{C}$ has a block structure: if $b_{ij} = 0$ then the corresponding block $b_{ij}\mathbf{C}$ will contain only zeros. The algorithm consists of a main procedure — *alternateLocalSearch* — and by three subroutines. The main procedure, detailed in Algorithm 2, swaps rows/columns and evaluates if the swaps lead to an improvement (reduction) in the value of function $\phi()$ (2). We employ two different metrics VAR and FROB to evaluate $\phi()$.

Algorithm 2: Alternate local search

Input: A, dimB, dimC
Output: solvedA, solvedB, solvedC
bestA = A;
metric = VAR;
A = kronGrouping(A);
blockMatrix, cornerizedMatrix = **false**;

/* Main loop   */
while $iter \leq maxiter$ **do**
    iter++;
    if $\sim blockMatrix$ **then**
        bestSwaps = outsiders(bestA);
    end
    if $\sim cornerizedMatrix \wedge blockMatrix$ **then**
        if $\sim balancedBlocks$ **then**
            /* Blocks have a different number of 1s, unfeasible factorization   */
            bestA = randPerm(bestA,75%);
            bestA = kronGrouping(bestA);
            blockMatrix = **false**;
            continue;
        end
        /* Matrix A has is a block matrix and the number of 1s is balanced, start the onionSearch
           subroutine   */
        [bestA, success] = onionSearch(bestA);
        if $success$ **then**
            break;
        else
            continue;
        end
    end
    swaps = [bestSwaps,baseSwaps];   // baseSwaps: all comb. of $\{1...dimA\}$ two at a time
    for $i=1$ **to** $len(swaps)$ **do**
        if $bestSwaps\ exhausted$ **then**
            baseSwaps = randPerm(baseSwaps,100%);
            if $metric == $ FROB **then**
                bestA = randPerm(bestA,45%);
                bestA = kronGrouping(bestA);
                blockMatrix = **false**;
                cornerizedMatrix = **false**;
            end
            if $metric == $ VAR **then**
                metric = FROB;
            end
        end
        curSwap = swaps(i);
        testA = swap(bestA, curSwap);
        if $improved$ **then**
            bestA = testA;
            if $metric == $ VAR $\wedge curVal \geq threshold$ **then**
                metric = FROB;
            end
            break;
        end
    end
    [solvedB, solvedC] = nearestKronProduct(bestA);
    if $\sim improved$ **then**
        nRestarts++;
        if $nRestarts$ mod $perturbateEvery == 0$ **then**
            /* Each perturbateEvery times the metric is swapped, A is completely perturbed   */
            bestA = randPerm(bestA,100%);
            bestA = kronGrouping(bestA);
            blockMatrix = **false**;
            cornerizedMatrix = **false**;
            metric = VAR;
        else
            swapMetric();
        end
    end
    if $nRestarts > maxRestarts * perturbateEvery \vee error < 0.000001$ **then**
        break;
    end
end
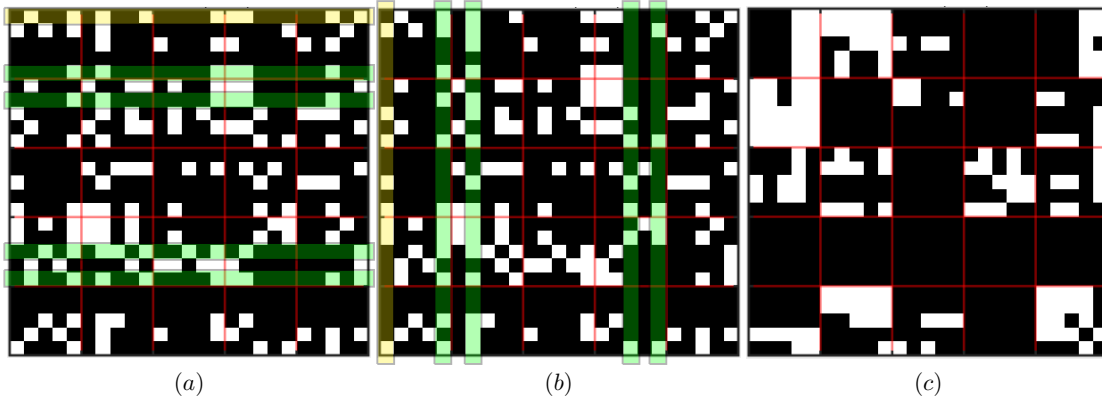**return** bestA, solvedB, solvedC;

Figure 2: *kronGrouping* procedure. In this example we show the rows/cols selection with $pivot = 1$ according to the best permutation found by the algorithm. (a) and (b): The current pivot (yellow) is compared with each other row and column of the matrix. (c) For each pivot, the $dimC - 1$ rows and columns that are most similar to it are selected as pivot neighbours in the current permutation (best viewed in color) and the permutation is finally applied. Es evident from (c), the resulting matrix is similar to a block matrix as desired. Please note that in this figure and the following ones, black squares stand for an element set to zero, and white squares stand for an element set to one.

Let s_{ij} be the number of 1s in the block $\mathbf{C}_{ij}$, and let VAR denote the variance of the number of 1s of each block, i.e., $\sigma^2 = Var\{s_{ij} \mid 1 \le i, j \le dimB\}$. If matrix $\mathbf{A}$ approaches *block matrix* form, then the variance decreases because, intuitively, $\mathbf{A}$ has "almost full" and "almost empty" blocks (refer to Definition 2.2). Indeed, a matrix already in block form is made of two types of blocks: *empty* blocks, that contain only 0s, and *full* blocks that are nonzero and share a considerable amount of 1s. Conversely, the metric FROB measures how much the nonempty blocks of $\mathbf{A}$ fit the Kronecker form, i.e., whether or not the content of the blocks is the same; this metric is based on the one used in [13] for solving the MKP problem: the rows of the same block of size $dimC \times dimC$ of $\mathbf{A}$ are concatenated, and become a row of a new matrix $\mathbf{F}$. $\mathbf{F}$ is then multiplied with $\mathbf{F}^\mathsf{T}$ element by element. Metric FROB is the squared sum of the elements of this product.

The local search procedure starts with metric VAR, and switches back and forth to FROB when no significant improvement in the value of $\phi()$ is observed for a given number of iterations. The use of two different metrics is motivated by the empirical observation that it speeds up convergence towards the solution, and helps to escape local minima. When we find a swap that improves the value of function $\phi()$, we apply the swap to the current permutation matrix $\mathbf{P}$ and proceed with the next iteration. The main loop terminates when the number of iterations exceeds a user-defined threshold, or when a decomposition is found.

Before swapping rows and columns, matrix $\mathbf{A}$ is processed by the *kronGrouping* procedure (Algorithm 3).

Algorithm 3: Kron grouping

Input: A
Output: groupedA
for *i=1* **to** *dimA* **do**
    for *j=1* **to** *dimA* **do**
        if *i==j* **then**
            MR(i,j)=MC(i,j)=0;
        else
            `/* Simil computes the similarity between two rows/cols using the dot`
            `    product.  To weight zero-based and one-based similarity, both A and 1-A`
            `    are considered.                                                    */`
            MR(i,j) = simil(A(i,:),A(j,:)) + simil(1-A(i,:),1-A(j,:));
            MC(i,j) = simil(A(:,i),A(:,j)) + simil(1-A(:,i),1-A(:,j));
        end
    end
end
`/* Six different permutations are tested, each one relying on a linear combination`
`    of rows similarity and columns similarity.                              */`
for *w=0* **to** *1* **step** *0.2* **do**
    MS = w * MR + (1-w) * MC;
    for *i=1* **to** *dimA* **do**
        pivot = first unused index;
        scan MS(pivot,:) for the (dimC-1) best indexes and add them to the current permutation;
    end
end
return best permutation according to metric = FROB

This procedure tries to permute **A** in such a way that it becomes similar to a block matrix. The intuition is that **A** should be permuted in such a way that it consists of blocks that contain as many 1s as possible, and others that are all zero. *kronGrouping* tries to exchange rows/columns in such a way that nonempty blocks (i.e., blocks of **A** that contain 1s) either lose or acquire 1s. To this end, *kronGrouping* evaluates the similarity between rows/columns; given two vectors $\mathbf{v} = (v_1, \ldots, v_n)$ and $\mathbf{w} = (w_i, \ldots, w_n)$, the similarity of $\mathbf{v}$ and $\mathbf{w}$ is a value that is proportional to the number of elements for which $v_i = w_i$. More specifically, as understandable from the last for loop of the algorithm, six different permutations are tested, each one relying on a linear combination of rows similarity and columns similarity. Refer to Figure 2 for an example. This information is used to produce a permutation that groups similar rows/columns in the same submatrix.

At each iteration of the main loop, *alternateLocalSearch* performs several operations before starting to test each possible swap. First of all, the procedure checks whether **A** is a block matrix. If not, subroutine *outsiders* is applied to **A** (Algorithm 4). While *kronGrouping* tries to rearrange the whole matrix at a glance, *outsiders* follows a fine tuning perspective and derives an ordered list of swaps that seem to be more convenient. This procedure allows to detect those rows and columns that are evidently misplaced with respect to the block structure we desire to reach.

To derive the swaps list, *outsiders* verifies which block of **A** should preferably be filled (i.e. it has a number of 1s above average) and which block should preferably be emptied. This condition always occurs when **A** is not a block matrix, as the number of elements set to one in nonempty blocks is unbalanced. The density of each block to be filled is considered as well: the best swap is supposed to be the one that moves 1s from a block to be emptied replacing zeros in a block to be filled. However, moving elements in blocks that already have a high density would produce overfilled blocks that are unlikely to appear in a feasible factorization.
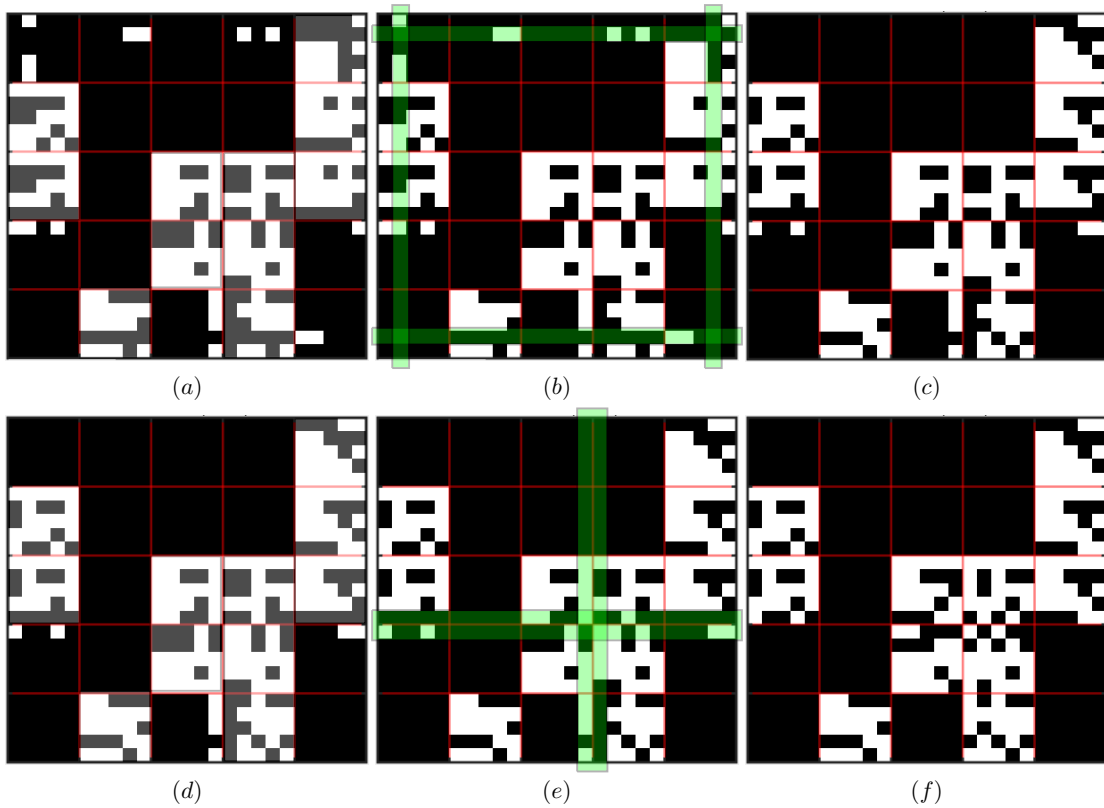
Figure 3: *outsiders* procedure. The procedure detects those blocks that should gain or lose 1s. In (a) blocks that are going to acquire 1s are highlighted. In (b) we show the best swap is among rows/columns 2 and 24. The main loop will use this swap as first, resulting in an improvement with respect to the current metric. As such, the swap is effectively performed (c). The function *outsiders* is called again at the next iteration and it selects $(15, 16)$ as best swap (e). The swap is then applied; the matrix is now divided in blocks (f).

Figure 4: *onionSearch* procedure, part 1. The binary block matrix EF (a) is permuted to maximize the dot product performed against the weight matrix W (b). When the local search reaches 75% of the optimum, the block matrix is deemed to be sufficiently rearranged in a top-left fashion and the procedure terminates (c). As evident from (b) –where onion layers' boundaries are highlighted in red– weights are set up to push the maximum number of filled blocks in those layers of the onion that are processed first.

Algorithm 4: Outsiders

Input: A
Output: bestSwaps
BL = blockMatrix(A);   // derive the dimB x dimB block matrix
S = blocksSum(BL);   // count the total number of 1s in each block
D = blocksDensity(BL);   // computes the density of 1s in each block
foreach *B* **in** *BL* **do**
   for *k=1* **to** *dimC* **do**
      sumRow = sum(B(k,:));
      sumCol = sum(B(:,k));
      if *S(B) < avg(S)* **then**
         /* The block is likely to be emptied, count the number of 1s   */
         WR(((row(B)-1)*dimC)+k,col(B)) = sumRow;
         WC(row(B),((col(B)-1)*app.dimC)+k) = sumCol;
      else
         /* The block is likely to be filled, count the number of zeros   */
         WR(((row(B)-1)*dimC)+k,col(B)) = -(dimC-sumRow);
         WC(row(B),((col(B)-1)*app.dimC)+k) = -(dimC-sumCol);
      end
   end
end
for *i=1* **to** *dimA* **do**
   p = blockIdx(i);
   for *j=1* **to** *dimA* **do**
      q = blockIdx(j);
      if *i and j fall in different blocks* **then**
         for *b=1* **to** *dimB* **do**
            if *BL(p,b) is a block to be emptied and BL(q,b) a block to be filled* **then**
               MS(i,j) = MS(i,j) + (WR(i,b) * WR(j,b) * $(1\text{-}D(q,b))^2$);
            end
            if *BL(p,b) is a block to be filled and BL(q,b) a block to be emptied* **then**
               MS(i,j) = MS(i,j) + (WR(i,b) * WR(j,b) * $(1\text{-}D(p,b))^2$);
            end
            if *BL(b,p) is a block to be emptied and BL(b,q) a block to be filled* **then**
               MS(i,j) = MS(i,j) + (WC(b,i) * WC(b,j) * $(1\text{-}D(b,q))^2$);
            end
            if *BL(b,p) is a block to be filled and BL(b,q) a block to be emptied* **then**
               MS(i,j) = MS(i,j) + (WC(b,i) * WC(b,j) * $(1\text{-}D(b,p))^2$);
            end
         end
      end
   end
end
if *sum(MS)==0* **then**
   blockMatrix = **true**;
   return null;
else
   return a list of swaps (i,j) in ascending order of MS(i,j);
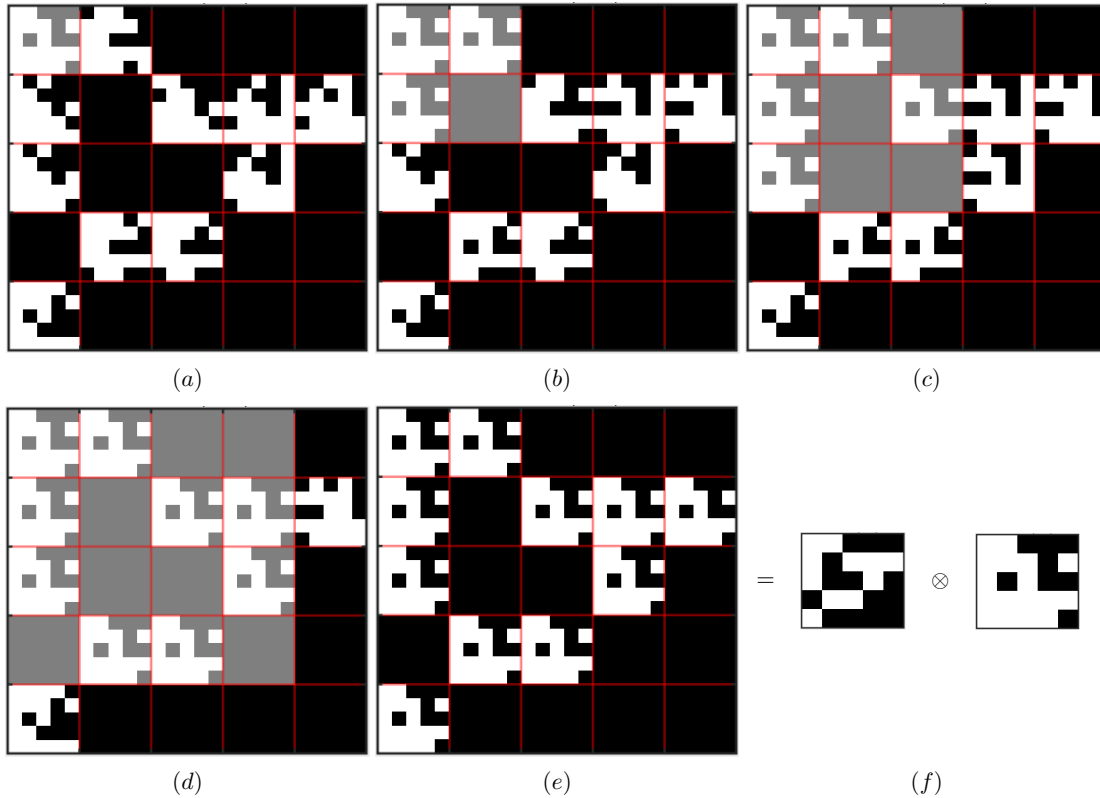**end**

Figure 5: *onionSearch* procedure, part 2. An example of the *onionSearch* procedure. The block matrix is factorized layer-by-layer. The first layer, composed by a single block $\mathbf{B}_{11}$, is implicitly factorized as $\mathbf{B}_{11} = \mathbf{I}_1 \otimes \mathbf{B}_{11}$ (a). It is thus used as template during the local search performed on the 2×2 blocks submatrix, composed of four blocks (b). As rows and columns swaps may only range in the second layer, the sole feasible solution is the one that permutes each block in accordance to the one serving as template. The procedure steps forward through the third (c), fourth (d) and fifth layer (e). As the local search on the last layer succeeds, the problem is globally solved. The solved factors are shown in (f).

When the procedure terminates the *bestSwaps* list is prepended to the randomized list of each legal swap (*baseSwaps*) that is used in the main loop as fallback. If the list of swaps is empty, there are no elements set to one in each of the blocks that are likely to be emptied. Thus, according to Definition 2.2, the matrix is now a block matrix. The basic principles of *outsiders* procedure are outlined in Figure 3. At the beginning, *outsiders* identifies which block of $\mathbf{A}$ should receive 1s (because it already has more 1s than average) and which block should lose 1s. Then, two matrices are produced, $\mathbf{WR}$ and $\mathbf{WC}$. $\mathbf{WR}$ contains the number of 1s found in each portion of a row of length $dimC$ — the portion of the row corresponding to a given block. If the block should receive 1s, the number of zeros is counted instead. The second matrix $\mathbf{WC}$ contains the same information computed column-wise. A third matrix $\mathbf{MS}$ is computed in order to find the best swap: the procedure loops on each row/column on a block-by-block basis and multiplies the elements of $\mathbf{WR}$ and $\mathbf{WC}$ corresponding to blocks that are in opposite conditions, i.e. one receives and the other loses 1s. The larger the values of $\mathbf{MW}$, the better.

This product is tuned according to the density of the block that receives 1s. However, moving

elements in blocks that already have a high number of 1s would produce blocks with too many 1s, which are unlikely to appear in a feasible factorization. Elements of **MS** are scanned and ordered to form the list of more promising swaps. If **MS** has zero elements only, there are no 1s on the blocks that should lose 1s. Therefore, according to Definition 2.2, the matrix is a block matrix.

Algorithm 5: Onion search

Input: A
Output: A
```
/* --- PART1:  rearrange the filled blocks in a top-left fashion ---           */
```
BL = blockMatrix(A);
EF = checkFilledBlocks(BL);   `// derive a boolean block matrix; 0:  empty block, 1:`
`o.w.`
W = weightMatrix(dimB);   `// the max weight corresponds to the top-left block`
cornerizeVal = sum(dotProduct(EF,W));
for *iter = 1* **to** *maxiterCornerize* **do**
    for *i=1* **to** *len(swaps)* **do**
        curSwap = swaps(i);
        testEF = swap(EF, curSwap);
        testVal = sum(dotProduct(testEF,W));
        if *testVal > cornerizeVal* **then**
            cornerizeVal = testVal;
            EF = testEF;
            BL = swap(BL, curSwap);
            break;
        end
    end
    if *cornerizeVal > 0.75 * cornerizeOptVal* **then**
        break;
    end
    if *∼improved* **then**
        BL = randPerm(BL,55%);
        Apply the same permutation to EF;
    end
end
cornerizedMatrix = **true**;
```
/* --- PART2:  perform a layer-by-layer Kron factorization on submatrices ---    */
```
A = fromBlocks(BL);
settledLayers = 1;
for *curlayer = 2* **to** *dimB* **do**
```
   /* In localSearchSubmatrix swaps are limited to the range of blocks
      [settledLayers+1, curLayer].  Moreover, only the Frobenius metric is used.
   */
```
    [success, A] = localSearchSubmatrix(A, curLayer, settledLayers);
    if *success* **then**
        settledLayers = curLayer;
    end
end
return success, A;

When **A** becomes a block matrix, *alternateLocalSearch* checks the number of 1s in each nonempty block. If the blocks have a different number of 1s, then matrix **A** is not (yet) in Kronecker form. If this happens, the local search got stuck in a (non optimum) local minimum; to escape the minimum, the program permutes 75%[2] of the rows/columns of **A** and the *alternateLocalSearch* procedure starts again. On the other hand, if the blocks of **A** do have the same number of 1s, then the matrix is in Kronecker form if and only if all blocks are the same. If the blocks are different,

---

[2]This value was determined empirically.

A is processed by the *onionSearch* subroutine.

The procedure *onionSearch* is divided in two parts (Algorithm 5). First, the block matrix is rearranged to push the maximum feasible number of blocks to the top-left corner, then, starting from the top-left corner, it performs a submatrix factorization in a layer-by-layer fashion. This second step is designed to adopt the first nonempty block as template and to reproduce the same layout in each other nonempty block.

Following the principles depicted in Figure 4, a binary block matrix is derived from **A** through permutations so that the dot product with a weight matrix is maximized. To find a convenient permutation, random swaps are applied following the same principle of the main local search. When the procedure reaches the 75% of the optimum (*cornerizeOptVal*), the *block matrix* is deemed to be sufficiently rearranged in a top-left fashion and the procedure terminates. Weights are defined to increase the maximum number of filled blocks in those layers of the onion that are processed first. We use the term "onion" to refer to the fact that the factorization works by examining matrix **A** layer-by-layer. When **A** reaches this form, we also refer to it as a *cornerizedMatrix* (see Algorithm 2 for reference).

We should now imagine the block matrix as being made of *dimB* layers, starting from the top-left corner. The first layer contains one block, the second layer contains three blocks, and so forth. The last layer is made of blocks that belong to the last row and column of the block matrix; refer to Figure 4b. The second part of *onionSearch* tries to find a feasible factorization for submatrices following a layer-by-layer perspective, as shown in Figure 5. The local search follows the same principles of *alternateLocalSearch* but uses metric FROB only. Moreover, rows/columns swaps are allowed in a limited range only. Specifically, swaps are limited to those indices corresponding to rows and columns that fall in the unsorted layers.

It is important to note that the single block $\mathbf{B}_{11}$ representing the first layer is implicitly factorized as $\mathbf{B}_{11} = \mathbf{I}_1 \otimes \mathbf{B}_{11}$. The special case where $\mathbf{B}_{11}$ is empty does not represent an issue. In this case, indeed, the procedure treats the first layer of the onion as already sorted and proceeds to the next layer. $\mathbf{B}_{11}$ is thus used as template during the local search performed on the successive 2×2 blocks submatrix, composed of four blocks. As rows and columns swaps may only range in the second — unsorted — layer, the sole feasible solution is the one that permutes each block in accordance to the one serving as template. If the local search performed on the 2×2 blocks submatrix succeeds, *onionSearch* steps to the third layer. Blocks falling in the first and in the second layer will stay untouched as they are sorted already. The procedure steps forward until the last layer is processed. If the local search on the last layer succeeds, the problem is globally solved. Otherwise, *alternateLocalSearch* proceeds testing swaps randomly to find a feasible solution.

Finally, to handle local minima, random permutations are applied in two more cases. First, it is possible that the list of best swaps returned by the *outsiders* procedure is exhausted without improvements. If this condition occurs through several iterations, a random permutation is applied to 45% of rows/columns of **A**. Second, if *alternateLocalSearch* fails to reduce the value of function $\phi()$ after a maximum number of iterations (i.e. both best swaps and default swaps are exhausted), a complete random permutation is applied (the local search restarts from scratch).

Note that, each time *alternateLocalSearch* applies a random permutation to escape from local minima, independently of its extent, the procedure *kronGrouping* is called again.
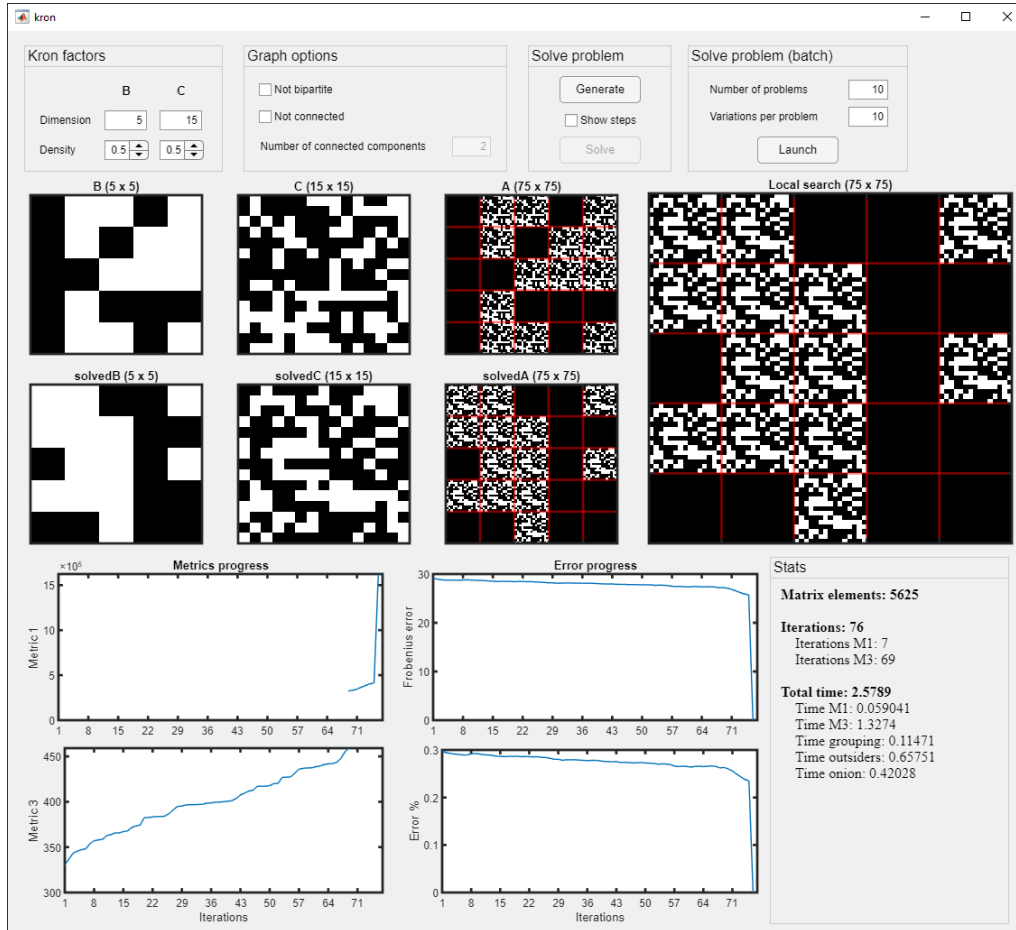
Figure 6: A sample snapshot of the MATLAB application implementing the heuristic.

4   Computational experiments

We implemented the heuristic described in Section 3 in the MATLAB programming environment; the source code is available online at https://github.com/calderonil/kron. The program provides a graphical user interface (Figure 6) that allows the user to generate a random binary matrix $\mathbf{A}$ of given size $dimA = dimB \times dimC$; the program then looks for a permutation $\mathbf{P}$ such that $\mathbf{A} = \mathbf{P}^\mathsf{T}(\mathbf{B} \otimes \mathbf{C})\mathbf{P}$ where $\mathbf{B}$ and $\mathbf{C}$ are binary matrices of size $dimB \times dimB$ and $dimC \times dimC$, respectively. $\mathbf{A}$ is generated starting from random $\mathbf{P}, \mathbf{B}, \mathbf{C}$, which are then "forgotten" and must therefore be computed from scratch; this ensures that a factorization of $\mathbf{A}$ always exists. The user must provide the densities ρ_B, ρ_C (fraction of 1s) of matrices $\mathbf{B}, \mathbf{C}$; the density ρ_A of $\mathbf{A}$ will therefore be $\rho_A = \rho_B \times \rho_C$. Although our application allows the user to impose additional constraints on the graph represented by the matrices (e.g., non-bipartitedness and/or non-connectedness), we did not impose any constraint for the experiments described in this section.

The combinations of parameters used in the experiments are shown in Table 2. The density (fraction of 1s) of $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are denoted as ρ_A, ρ_B and ρ_C, respectively. We consider several

ρ_B, ρ_C, ρ_A	Size of B	α	Size of C	Size of A	failure %	t_{min}	t'_{avg}	t_{avg}	t_{max}
		1	5×5	25×25	0.00%	0.02	0.03	0.06	0.31
	5×5	2	10×10	50×50	0.00%	0.07	0.09	0.13	0.82
		3	15×15	75×75	0.00%	0.18	0.21	0.27	0.81
		1	7×7	49×49	0.00%	0.08	0.11	0.24	1.28
0.5, 0.5, 0.25	7×7	2	14×14	98×98	0.00%	0.35	0.43	0.67	2.55
		3	21×21	147×147	1.00%	1.00	1.30	2.85	35.72
		1	10×10	100×100	0.00%	0.52	0.59	1.42	11.59
	10×10	2	20×20	200×200	0.00%	2.50	2.77	3.49	16.57
		3	30×30	300×300	0.00%	6.71	7.54	8.47	10.89
		1	5×5	25×25	0.00%	0.01	0.03	0.06	0.52
	5×5	2	10×10	50×50	0.00%	0.06	0.08	0.24	3.14
		3	15×15	75×75	0.00%	0.16	0.19	0.27	1.41
		1	7×7	49×49	0.00%	0.07	0.12	0.45	2.33
0.6, 0.6, 0.36	7×7	2	14×14	98×98	3.00%	0.39	0.45	1.20	15.30
		3	21×21	147×147	0.00%	0.90	1.05	1.80	35.97
		1	10×10	100×100	0.00%	0.36	0.51	0.85	8.42
	10×10	2	20×20	200×200	0.00%	2.14	2.67	7.14	87.05
		3	30×30	300×300	0.00%	6.58	7.98	16.79	210.95
		1	5×5	25×25	1.00%	0.01	0.04	0.09	1.06
	5×5	2	10×10	50×50	1.00%	0.07	0.17	0.63	6.22
		3	15×15	75×75	4.00%	0.14	0.34	0.84	10.32
		1	7×7	49×49	0.00%	0.06	0.18	0.50	3.21
0.7, 0.7, 0.49	7×7	2	14×14	98×98	7.00%	0.32	1.18	1.97	18.71
		3	21×21	147×147	9.00%	0.88	6.21	8.56	102.26
		1	10×10	100×100	0.00%	0.25	0.45	1.84	19.42
	10×10	2	20×20	200×200	9.00%	2.16	5.84	14.91	128.62
		3	30×30	300×300	0.00%	5.84	6.65	19.91	243.12

Table 2: Each row summarizes the performance of our heuristic on 10 different initial random matrices $\mathbf{A}$; for each matrix $\mathbf{A}$ we executed the heuristic 10 times, each one starting from a random permutation of $\mathbf{A}$. $\alpha = dimC/dimB$ is the ratio between the sizes of $\mathbf{C}$ and $\mathbf{B}$. t'_{avg} is the average of the minimum execution times of each different problem. The time spent on instances for which no solution has been found have been excluded from average, minimum and maximum execution times.

combinations of sizes $dimB, dimC$ according to a parameter α that denotes the ratio between the size of $\mathbf{C}$ and $\mathbf{B}$ ($\alpha = dimC/dimB$). Intuitively, high values of α denote that the size of the factors of $\mathbf{A}$ are different. As this assumption does not cause a loss of generality, we also set $dimC \geq dimB$. As $dimB$ also defines the number of blocks of $\mathbf{A}$, limiting this parameter improves the algorithm speed, as many procedures loop through the blocks of $\mathbf{A}$.

Each row of the table summarizes the result of one *run*: a run consists of 10 random problem instances with the chosen parameters. Since our heuristic is sensitive to the initial conditions, for each instance we consider 10 initial random permutations of the matrix $\mathbf{A}$; this is equivalent to fixing $\mathbf{B}, \mathbf{C}$ and choosing ten different random permutation matrices $\mathbf{P}$. Therefore, each row summarizes 10×10 executions of our program.

The program has been executed on a desktop PC with an Intel Xeon CPU running at 3.30 GHz with 16 GB of RAM running Windows 10 (Matlab R2021a). Both a *single instance mode* and a *batch mode* are provided; the former solves a single problem, while the latter generates a set of random instances with the same parameters ($dimA, dimB, dimC$). To collect the results we show in the following, the program was executed in batch mode.

ρ_B, ρ_C, ρ_A	Size of B	α	Size of C	Size of A	failure %	t_{min}	t'_{avg}	t_{avg}	t_{max}
	5×5	8	40×40	200×200	0.00%	1.08	1.31	5.15	22.38
	10×10	2	20×20	200×200	0.00%	0.97	1.09	1.19	3.01
	14×14	1	14×14	196×196	0.00%	0.98	1.16	1.48	3.35
	5×5	12	60×60	300×300	0.00%	4.36	5.05	12.45	67.66
	10×10	3	30×30	300×300	0.00%	3.15	3.37	3.70	4.48
	17×17	1	17×17	289×289	0.00%	4.16	4.39	5.34	9.90
	5×5	16	80×80	400×400	0.00%	40.18	40.85	44.29	52.01
0.5, 0.5, 0.25	10×10	4	40×40	400×400	0.00%	11.59	12.09	13.12	14.76
	20×20	1	20×20	400×400	0.00%	14.41	14.52	16.23	18.62
	5×5	20	100×100	500×500	0.00%	59.95	63.83	65.69	68.37
	10×10	5	50×50	500×500	0.00%	27.79	29.09	30.95	35.36
	22×22	1	22×22	484×484	0.00%	24.47	25.33	26.30	27.40
	5×5	25	125×125	625×625	0.00%	154.32	155.68	165.12	177.86
	10×10	6.2	62×62	620×620	0.00%	81.20	81.96	141.13	304.69
	25×25	1	25×25	625×625	0.00%	66.46	66.70	70.73	80.20

Table 3: This table shows another set of experiments conducted on bigger matrices. Each row summarizes the performance of the heuristic on 10 different initial random matrices $\mathbf{A}$; for each matrix $\mathbf{A}$ we executed the heuristic 10 times, each one starting from a random permutation of $\mathbf{A}$. Five groups of experiments are listed, ranging from 200×200 matrices to 625×625 matrices, with respect to the size of $\mathbf{A}$. For each group of experiments, the size of $\mathbf{A}$ was fixed, varying the ratio α. Each group includes the $\alpha = 1$ case, i.e., the case where the size of $\mathbf{B}$ and $\mathbf{C}$ is the same.

For a single instance, our program executes *alternateLocalSearch* for at most 500 iterations (see the main loop of Algorithm 2); when no feasible factorization is found after the maximum allowed number of iterations, the procedure stops and the failure count is incremented.

For each run we collect five metrics: the percentage of executions at the end of which no factorization was found (failure %); the minimum and maximum execution times in seconds (t_{min} and t_{max}, respectively); the average execution time across all executions that *did* find the optimal solution (t_{avg}); the average of the minimum execution time for each different problem (t'_{avg}). More precisely, t'_{avg} is computed as follows:

1. For each combination of parameters, we generate 10 random instances consisting of matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ of the appropriate size and content;

2. For each instance, we generate 10 random permutation matrices $\mathbf{P}$; we get 10 variations $\mathbf{A} = \mathbf{P}^\mathsf{T}(\mathbf{B} \otimes \mathbf{C})\mathbf{P}$ that only differ by the permutation $\mathbf{P}$

3. We compute the minimum time to get the solution of the 10 variations from the previous step;

4. t'_{avg} is the average of the minimum times computed at the previous step.

t'_{avg} is useful because there is a significant variance across the execution times of the 10 variations of the same problem (see below). Indeed, as stated above, there might be a huge variation in the time required to factorize a matrix $\mathbf{A} = \mathbf{P}^\mathsf{T}(\mathbf{B} \otimes \mathbf{C})\mathbf{P}$ depending on the permutation $\mathbf{P}$. Therefore, t'_{avg} is the average time to solve a problem instance if we were able to parallelize the heuristic across 10 independent execution units, so that the first one that gets the solution stops the computation.

As can be observed from Table 2, the minimum time to compute a solution is very low (a few seconds) for all problems. The largest matrix $\mathbf{A}$ (size 300×300) can be factored in $t_{min} = 6.71s$. It should be observed that as the density of $\mathbf{A}$ increases, the problem becomes more difficult for
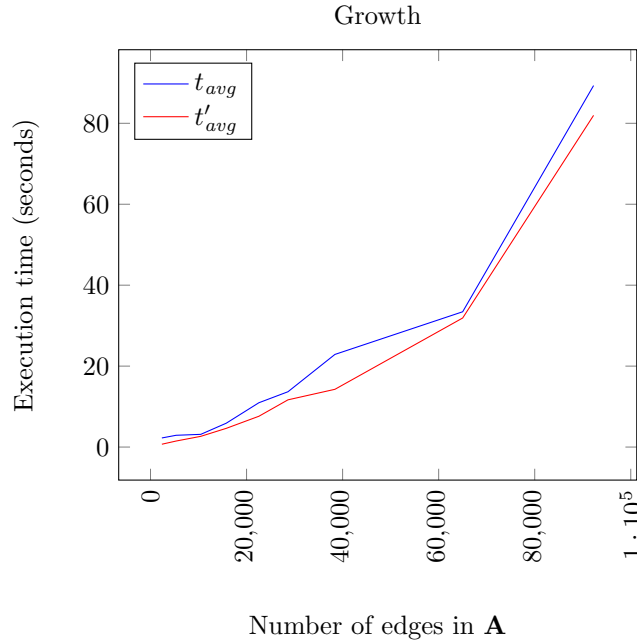
Figure 7: Dependence of the execution time on the number of 1s of $\mathbf{A}$. Given $\rho_A = 0.25$ and $dimB = 10$, this chart shows the linear growth for t_{avg} and t'_{avg} depending on the size of $\mathbf{C}$. Showing the matrix dimension in the x-axis could be misleading, the number of edges in $\mathbf{A}$ is used instead.

our heuristic as witnessed by the increasing fraction of unsolved instances. We also observe that there is a large variability between the minimum and maximum times required to solve an instance; indeed we observe that the gap between t_{min} and t_{max} becomes more than an order of magnitude, especially for large matrices $\mathbf{A}$ that are decomposed into factors of unbalanced sizes ($\alpha = 3$). This is due to the fact that the heuristic is sensitive to both the permutation $\mathbf{P}$, and to the sequence of swaps that are applied during the computation (the swaps are in part generated pseudo-randomly).

To strengthen the study, we also carried out a set of tests on wider matrices, following the very same approach. In this second set of tests, we kept the density fixed. Five groups of experiments were considered, ranging from 200×200 matrices to 625×625 matrices, with respect to the size of $\mathbf{A}$. For each group of experiments, the size of $\mathbf{A}$ was fixed, varying the ratio α. Experiment results are listed in Table 3.

We now turn our attention to the study of the dependence of the execution time on the number of edges of the graph whose adjacency matrix is $\mathbf{A}$; the number of edges is simply the number of 1s in $\mathbf{A}$. To this aim, we performed 90 additional experiments (9 separate problem instances with 10 random initial permutations each). We set $\rho_A = 0.25$ and $dimB = 10$, and we increased the dimension of $\mathbf{C}$ according to the following steps: $dimC = \{10, 15, 20, 25, 30, 35, 40, 50, 62\}$. As shown in Figure 7, t_{avg} and t'_{avg} grow more or less linearly with respect to the number of edges of $\mathbf{A}$ until $dimC$ reaches the value of 50 (i.e., until $\mathbf{A}$ has approximately 65000 edges). When this value is exceeded, the execution time increases at a faster rate.

5   Conclusions and Future Works

In this paper we presented a heuristic for decomposing a directed graph into factors according to the direct product: given a directed, unweighted graph G with adjacency matrix $\mathbf{Adj}(G)$, our heuristic searches for a pair of graphs G_1 and G_2 such that $G = G_1 \times G_2$, where $G_1 \times G_2$ is the direct product of G_1 and G_2. The heuristic proposed in this paper represents – to the best of our knowledge – the first computational approach for general directed, unweighted graphs. We provided a MATLAB implementation that we used to run a set of computational experiments to assess the effectiveness of our approach. Our implementation can factorize a graph of size 300×300 in a few seconds. In a few worst-case scenarios the time grows to a few minutes, and is due to the fact that our heuristic is sensitive to the structure of the input; although it may fail to find a solution, in our experiments we observed failures in just a few instances. The complete source code of the Matlab implementation can be downloaded from `https://github.com/calderonil/kron`.

We are planning to extend the heuristic along two directions: first, to handle *weighted* graphs instead of just unweighted ones; second, to compute the *approximate* Kronecker decomposition of unweighted graphs, where (a suitable permutation of) the input matrix $\mathbf{A}$ can be expressed as a Kronecker product of two smaller matrices $\mathbf{B}$ and $\mathbf{C}$, plus an additional binary error term $\mathbf{E}$, i.e., $\mathbf{A} = \mathbf{P}^\mathsf{T}(\mathbf{B} \otimes \mathbf{C})\mathbf{P} + \mathbf{E}$.

Acknowledgements

References

[1] N. Alon and E. Lubetzky. Independent sets in tensor graph powers. *Journal of Graph Theory*, 54(1):73–87, 2007. `doi:10.1002/jgt.20194`.

[2] H. Arndt. Load balancing: dimension exchange on product graphs. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 20–, 2004. `doi:10.1109/IPDPS.2004.1302928`.

[3] L. Calderoni, L. Margara, and M. Marzolla. Direct product primality testing of graphs is gi-hard. *Theoretical Computer Science*, 860:72–83, 2021. `doi:10.1016/j.tcs.2021.01.029`.

[4] V. Diekert and J. Kausch. Logspace computations in graph products. *Journal of Symbolic Computation*, 75:94–109, 2016. Special issue on the conference ISSAC 2014: Symbolic computation and computer algebra. `doi:10.1016/j.jsc.2015.11.009`.

[5] S. Ghozati. A finite automata approach to modeling the cross product of interconnection networks. *Mathematical and Computer Modelling*, 30(7):185–200, 1999. `doi:10.1016/S0895-7177(99)00173-9`.

[6] R. Hammack, W. Imrich, and S. Klavžar. *Handbook of Product Graphs, Second Edition*. Discrete Mathematics and Its Applications. Taylor & Francis, 2011.

[7] W. Imrich. Factoring cardinal product graphs in polynomial time. *Discrete Mathematics*, 192(1):119–144, 1998. `doi:10.1016/S0012-365X(98)00069-7`.

[8] W. Imrich and D. Rall. Finite and infinite hypercubes as direct products. *Australasian journal of combinatorics*, 36:83, 2006.

[9] H. Z. Jooya, K. Reihani, and S.-I. Chu. A graph-theoretical representation of multiphoton resonance processes in superconducting quantum circuits. *Scientific Reports*, 6, 2016. `doi:10.1038/srep37544`.

[10] R. Lammprey and B. Barnes. Products of graphs and applications. *Model. Simul.*, 5:1119–1123, 1974.

[11] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11(33):985–1042, 2010. URL: `http://jmlr.org/papers/v11/leskovec10a.html`.

[12] C. F. Loan. The ubiquitous Kronecker product. *Journal of Computational and Applied Mathematics*, 123(1):85–100, 2000. Numerical Analysis 2000. Vol. III: Linear Algebra. `doi:10.1016/S0377-0427(00)00393-9`.

[13] C. F. Van Loan and N. Pitsianis. Approximation with Kronecker products. In M. S. Moonen, G. H. Golub, and B. L. R. De Moor, editors, *Linear Algebra for Large Scale and Real-Time Applications*, pages 293–314. Springer Netherlands, Dordrecht, 1993. `doi:10.1007/978-94-015-8196-7_17`.

[14] P. M. Weichsel. The kronecker product of graphs. *Proceedings of the American Mathematical Society*, 13(1):47–52, 1962.