Partitioning, matching, and ordering: Combinatorial scientific computing with matrices and tensors

Bora Uçar

MÉMOIRE D'HABILITATION À DIRIGER DES RECHERCHES

présenté le 19 septembre 2019

à l'École Normale Supérieure de Lyon

par

Bora Uçar

Partitioning, matching, and ordering: Combinatorial scientific computing with matrices and tensors

(Partitionnement, couplage et permutation: Calcul scientifique combinatoire sur des matrices et des tenseurs)

Devant le jury composé de :

Rob H. Bisseling	*Utrecht University, the Netherlands*	Rapporteur
Nadia Brauner	*Université Joseph Fourier, Grenoble France*	Examinatrice
Karen D. Devine	*Sandia National Labs, Albuquerque, New Mexico, USA*	Rapporteuse
Ali Pınar	*Sandia National Labs, Livermore, California, USA*	Examinateur
Yves Robert	*ENS Lyon, France*	Examinateur
Denis Trystram	*Grenoble INP, France*	Rapporteur

Contents

List of Algorithms

Chapter 1

Introduction

This document investigates three classes of problems at the interplay of discrete algorithms, combinatorial optimization, and numerical methods. The general research area is called combinatorial scientific computing (CSC). In CSC, the contributions have practical and theoretical flavor. For all problems discussed in this document, we have the design, analysis, and implementation of algorithms along with many experiments. The theoretical results are included in this document, some with proofs; the reader is invited to the original papers for the omitted proofs. A similar approach is taken for presenting the experiments. While most results for observing theoretical findings in practice are included, the reader is referred to the original papers for some other results (e.g., run time analysis).

The three problem classes are that of partitioning, matching, and ordering. We cover two problems from the partitioning class, three from the matching class, and two from the ordering class. Below, we present these seven problems after introducing the notation and recalling the related definitions. We summarize the computing contexts in which problems arise and highlight our contributions.

1.1 Directed graphs

A *directed graph* $G = (V, E)$ is a set V of vertices and a set E of directed edges of the form $e = (u, v)$, where e is directed from u to v. A *path* is a sequence of edges $(u_1, v_1) \cdot (u_2, v_2) \cdots$ with $v_i = u_{i+1}$. A path $((u_1, v_1) \cdot (u_2, v_2) \cdots (u_\ell, v_\ell))$ is of length ℓ, where it connects a sequence of $\ell+1$ vertices $(u_1, v_1 = u_2, \ldots, v_{\ell-1} = u_\ell, v_\ell)$. A path is called *simple* if the connected vertices are distinct. Let $u \rightsquigarrow v$ denote a simple path that starts from u and ends at v. We say that a vertex v is *reachable* from another vertex u, if a path connects them. A path $((u_1, v_1) \cdot (u_2, v_2) \cdots (u_\ell, v_\ell))$ forms a (simple) *cycle* if all v_i for $1 \leq i \leq \ell$ are distinct and $u_1 = v_\ell$. A *directed acyclic graph*, DAG in short, is a directed graph with no cycles. A directed graph

is *strongly connected*, if every vertex is reachable from every other vertex.

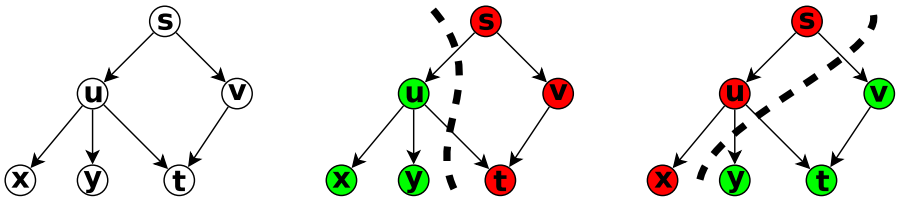
A directed graph $G' = (V', E')$ is said to be a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. We call G' as the subgraph of G *induced* by V', denoted as $G[V']$, if $E' = E \cap (V' \times V')$. For a vertex set $S \subseteq V$, we define $G - S$ as the subgraph of G induced by $V - S$, i.e., $G[V - S]$. A vertex set $C \subseteq V$ is called a *strongly connected component* of G, if the induced subgraph $G[C]$ is strongly connected, and C is maximal in the sense that for all $C' \supset C$, $G[C']$ is not strongly connected. A transitive reduction G^0 is a minimal subgraph of G such that if u is connected to v in G then u is connected to v in G^0 as well. If G is acyclic, then its transitive reduction is unique.

The path $u \rightsquigarrow v$ represents a dependency of v to u. We say that the edge (u, v) is *redundant* if there exists another $u \rightsquigarrow v$ path in the graph. That is, when we remove a redundant (u, v) edge, u remains connected to v, and hence, the dependency information is preserved. We use $\texttt{Pred}[v] = \{u \mid (u, v) \in E\}$ to represent the (immediate) predecessors of a vertex v, and $\texttt{Succ}[v] = \{u \mid (v, u) \in E\}$ to represent the (immediate) successors of v. We call the neighbors of a vertex v, its immediate predecessors and immediate successors: $\texttt{Neigh}[v] = \texttt{Pred}[v] \cup \texttt{Succ}[v]$. For a vertex u, the set of vertices v such that $u \rightsquigarrow v$ are called the *descendants* of u. Similarly, the set of vertices v such that $v \rightsquigarrow u$ are called the *ancestors* of the vertex u. The vertices without any predecessors (and hence ancestors) are called the *sources* of G, and vertices without any successors (and hence descendants) are called the *targets* of G. Every vertex u has a weight denoted by w_u and every edge $(u, v) \in E$ has a cost denoted by $c_{u,v}$.

A k-way partitioning of a graph $G = (V, E)$ divides V into k disjoint subsets $\{V_1, \ldots, V_k\}$. The weight of a part V_i denoted by $w(V_i)$ is equal to $\sum_{u \in V_i} w_u$, which is the total vertex weight in V_i. Given a partition, an edge is called a *cut edge* if its endpoints are in different parts. The *edge cut* of a partition is defined as the sum of the costs of the cut edges. Usually, a constraint on the part weights accompanies the problem. We are interested in acyclic partitions, which are defined below.

Definition 1.1 (**Acyclic k-way partition).** A partition $\{V_1, \ldots, V_k\}$ of $G = (V, E)$ is called an acyclic k-way partition if two paths $u \rightsquigarrow v$ and $v' \rightsquigarrow u'$ do not co-exist for $u, u' \in V_i$, $v, v' \in V_j$, and $1 \leq i \neq j \leq k$.

In other words, for a suitable numbering of the parts, all edges should be directed from a vertex in a part p to another vertex in a part q where $p \leq q$. Let us consider a toy example shown in Fig. 1.1a. A partition of the vertices of this graph is shown in Fig. 1.1b with a dashed curve. Since there is a cut edge from s to u and another from u to t, the partition is cyclic. An acyclic partition is shown in Fig. 1.1c, where all the cut edges are from one part to the other.



(a) A toy graph

(b) A partition ignoring the directions; it is cyclic.

(c) An acyclic partitioning.

Figure 1.1: A toy graph with six vertices and six edges, a cyclic partitioning, and an acyclic partitioning.

There is a related notion in the literature [87], which is called a convex partition. A partition is convex if for all vertex pairs u, v in the same part, the vertices in any $u \rightsquigarrow v$ path are also in the same part. Hence, if a partition is acyclic it is also convex. On the other hand, convexity does not imply acyclicity. Figure 1.2 shows that the definitions of an acyclic partition and a convex partition are not equivalent. For the toy graph in Fig. 1.2a, there are three possible balanced partitions shown in Figures 1.2b, 1.2c, and 1.2d. They are all convex, but only the one in Fig. 1.2d is acyclic.

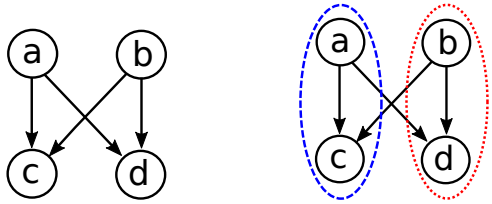
Deciding on the existence of a k-way acyclic partition respecting an upper bound on the part weights and an upper bound on the cost of cut edges is NP-complete [93, ND15]. In Chapter 2, we treat this problem which is formalized as follows.

Problem 1 (DAGP: DAG partitioning)**.** *Given a directed acyclic graph $G = (V, E)$ with weights on vertices, an imbalance parameter ε, find an acyclic k-way partition $\{V_1, \ldots, V_k\}$ of V such that the balance constraints*

$$w(V_i) \leq (1 + \varepsilon)\frac{w(V)}{k}$$

are satisfied for $1 \leq i \leq k$, and the edge cut, which is equal to the number of edges whose end points are in two different parts, is minimized.

The DAGP problem arises in exposing parallelism in automatic differentiation [53, Ch.9], and particularly in the computation of the Newton step for solving nonlinear systems [51, 52]. The DAGP problem with some additional constraints is used to reason about the parallel data movement complexity and to dynamically analyze the data locality potential [84, 87]. Other important applications of the DAGP problem include (i) fusing loops for improving temporal locality, and enabling streaming and array contractions in runtime systems [144, 145]; (ii) analysis of cache efficient execution



(a) A toy graph

(b) A cyclic and convex partition

(c) A cyclic and convex partition

(d) An acyclic and convex partition

Figure 1.2: A toy graph, two cyclic and convex partitions, and an acyclic and convex partition.

of streaming applications on uniprocessors [4]; (iii) a number of circuit design applications in which the signal directions impose acyclic partitioning requirement [54, 213].

In Chapter 2, we address the DAGP problem. We adopt the multilevel approach [35, 113] for this problem. This approach is the de-facto standard for solving graph and hypergraph partitioning problems efficiently, and used by almost all current state-of-the-art partitioning tools [27, 40, 113, 130, 186, 199]. The algorithms following this approach have three phases: the coarsening phase, which reduces the number of vertices by clustering them; the initial partitioning phase, which finds a partition of the coarsest graph; and the uncoarsening phase, in which the initial partition is projected to the finer graphs and refined along the way, until a solution for the original graph is obtained. We focus on two-way partitioning (sometimes called bisection), as this scheme can be used in a recursive way for multiway partitioning. To ensure the acyclicity of the partition at all times, we propose novel and efficient coarsening and refinement heuristics. We also propose effective ways to use the standard undirected graph partitioning methods in our multilevel scheme. We perform a large set of experiments on a dataset and report significant improvements compared to the current state of the art.

A *rooted tree* T is an ordered triple $(\mathcal{V}, r, \rho)$ where $\mathcal{V}$ is the vertex set, $r \in \mathcal{V}$ is the root vertex, and ρ gives the immediate ancestor/descendant relation. For two vertices v_i, v_j such that $\rho(v_j) = v_i$, we call v_i the *parent*

vertex of v_j, and call v_j a *child* vertex of v_i. With this definition the root r is the ancestor of every other node in T. The children vertices of a vertex are called *sibling* vertices to each other. The *height* of T, denoted as $h(T)$, is the number of vertices in the longest path from a leaf to the root r.

Let $G = (V, E)$ be a directed graph with n vertices which are ordered (or labeled as $1, \ldots, n$). The *elimination tree* of a directed graph is defined as follows [81]. For any $i \neq n$, $\rho(i) = j$ whenever $j > i$ is the minimum vertex id such that the vertices i and j lie in the same strongly connected component of the subgraph $G[\{1, \ldots, j\}]$—that is i and j are in the same strongly connected component of the subgraph containing the first j vertices, and j is the smallest number. Currently the algorithm with the best worst-case time complexity has a run time of $O(m \log n)$ [133]—another algorithm with the worst-case time complexity of $O(mn)$ [83] also performs well in practice. As the definition of the elimination tree implies, the properties of the tree depend on the ordering of the graph.

The elimination tree is a recent model playing important roles in sparse LU factorization. This tree captures the dependencies between the tasks of some well-known variants of sparse LU factorization (we review these in Chapter 7). Therefore, the height of the elimination tree corresponds to the critical path length of the task dependency graph in the corresponding parallel LU factorization methods. In Chapter 7, we investigate the problem of finding minimum height elimination trees, formally defined below, to expose a maximum degree of parallelism by minimizing the critical path length.

Problem 2 (Minimum height elimination tree)**.** *Given a directed graph, find an ordering of its vertices so that the associated elimination tree has the minimum height.*

The minimum height of an elimination tree of a given directed graph corresponds to a directed graph parameter known as the cycle-rank [80]. The problem being NP-complete [101], we propose heuristics in Chapter 7, which generalize the most successful approaches used for symmetric matrices to unsymmetric ones.

1.2 Undirected graphs

An undirected graph $G = (V, E)$ is a set V of vertices and a set E of edges. Bipartite graphs are undirected graphs, where the vertex set can be partitioned into two sets with all edges connecting vertices in the two parts. Formally, $G = (V_R \cup V_C, E)$ is a bipartite graph, where $V = V_R \cup V_C$ is the vertex set, and for each edge $e = (r, c)$ we have $r \in V_R$ and $c \in V_C$. The number of edges incident on a vertex is called its degree. A *path* in a graph is a sequence of vertices such that each consecutive vertex pair share

an edge. A vertex *is reachable from* another one, if there is a path between them. The *connected components* of a graph are the equivalence classes of vertices under the "is reachable from" relation. A *cycle* in a graph is a path whose start and end vertices are the same. A *simple cycle* is a cycle with no vertex repetitions. A *tree* is a connected graph with no cycles. A *spanning tree* of a connected graph G is a tree containing all vertices of G.

A matching $\mathcal{M}$ in a graph $G = (V, E)$ is a subset of edges E where a vertex in V is in at most one edge in $\mathcal{M}$. Given a matching $\mathcal{M}$, a vertex v is said to be *matched* by $\mathcal{M}$ if v is in an edge of $\mathcal{M}$, otherwise v is called *unmatched*. If all the vertices are matched by $\mathcal{M}$, then $\mathcal{M}$ is said to be a *perfect matching*. The *cardinality* of a matching $\mathcal{M}$, denoted by $|\mathcal{M}|$, is the number of edges in $\mathcal{M}$. In Chapter 4, we treat the following problem, which asks to find approximate matchings.

Problem 3 (Approximating the maximum cardinality of a matching in undirected graphs)**.** *Given an undirected graph $G = (V, E)$, design a fast (near linear time) heuristic to obtain a matching $\mathcal{M}$ of large cardinality with an approximation guarantee.*

There are a number of polynomial time algorithms to solve the maximum cardinality matching problem exactly in graphs. The lowest worst-case time complexity of the known algorithms is $\mathcal{O}(\sqrt{n}\tau)$ for a graph with n vertices and τ edges. For bipartite graphs, the first of such algorithms is described by Hopcroft and Karp [117]; certain variants of the push-relabel algorithm [96] also have the same complexity (we give a classification for maximum cardinality matchings in bipartite graphs in a recent paper [132]). For general graphs, algorithms with the same complexity are known [30, 92, 174]. There is considerable interest in simpler and faster algorithms that have some approximation guarantee [148]. Such algorithms are used as a jump-start routine by the current state of the art exact maximum matching algorithms [71, 148, 169]. Furthermore, there are applications [171] where large cardinality matchings are used.

The specialization of Problem 3 for bipartite graphs and general undirected graphs are described in, respectively, Section 4.2 and Section 4.3. Our focus is on approximation algorithms which expose parallelism and have linear run time in practical settings. We propose randomized heuristics and analyze their results. The proposed heuristics construct a subgraph of the input graph by randomly choosing some edges. They then obtain a maximum cardinality matching on the subgraph and return it as an approximate matching for the input graph. The probability density function for choosing a given edge is obtained by scaling a suitable matrix to be doubly stochastic. The heuristics for the bipartite graphs and general undirected graphs share this general approach and differ in the analysis. To the best of our knowledge, the proposed heuristics have the highest constant factor approximation

ratio (around 0.86 for large graphs).

1.3 Hypergraphs

A hypergraph $H = (V, E)$ is defined as a set of vertices V and a set of hyperedges E. Each hyperedge is a set of vertices. A matching in a hypergraph is a set of disjoint hyperedges.

A hypergraph $H = (V, E)$ is called *d-partite* and *d-uniform*, if $V = \bigcup_{i=1}^{d} V_i$ with disjoint V_is and every hyperedge contains a single vertex from each V_i. In Chapter 6, we investigate the problem of finding matchings in hypergraphs, formally defined as follows.

Problem 4 (Maximum matching in d-partite d-uniform hypergraphs)**.** *Given a d-partite d-uniform hypergraph, find a matching of maximum size.*

The problem is NP-complete for $d \geq 3$; the 3-partite case is called the Max-3-DM problem [126], and is an important problem in combinatorial optimization. It is a special case of the d-Set-Packing problem [110]. It has been shown that d-Set-Packing is hard to approximate within a factor of $\mathcal{O}(d/\log d)$ [110]. The maximum/perfect set packing problem has many applications, including combinatorial auctions [98] and personnel scheduling [91]. Matchings in hypergraphs can also be used in the coarsening phase of multilevel hypergraph partitioning tools [40]. In particular, d-uniform and d-partite hypergraphs arise in modeling tensors [134] and heuristics for Problem 4 can be used to partition these hypergraphs, by plugging them in the current state of the art partitioning tools. We propose heuristics for Problem 4. We first generalize some graph matching heuristics, then propose a novel heuristic based on tensor scaling to improve the quality via judicious hyperedge selections. Experiments on random, synthetic and real-life hypergraphs show that this new heuristic is highly practical and superior to the others on finding a matching with large cardinality.

1.4 Sparse matrices

Bold, upper case Roman letters are used for matrices, as in $\mathbf{A}$. Matrix elements are shown with the corresponding lowercase letters, for example $a_{i,j}$. Matrix sizes are sometimes shown in the lower right corner, e.g., $\mathbf{A}_{I \times J}$. Matlab notation is used to refer to the entire rows and columns of a matrix, e.g., $\mathbf{A}(i,:)$ and $\mathbf{A}(:,j)$ refer to the ith row and jth column of $\mathbf{A}$ respectively. A permutation matrix is a square, $\{0,1\}$ matrix, with exactly one nonzero in each row and each column. A sub-permutation matrix is a square, $\{0,1\}$ matrix, with at most one nonzero in a row or a column.

A graph $G = (V, E)$ can be represented as a sparse matrix $\mathbf{A}_G$ called the adjacency matrix. In this matrix an entry $a_{ij} = 1$ if the vertices v_i and v_j in G are incident on a common edge, and $a_{ij} = 0$ otherwise. If G is bipartite with $V = V_R \cup V_C$, we identify each vertex in V_R with a unique row of $\mathbf{A}_G$, and each vertex in V_C with a unique column of $\mathbf{A}_G$. If G is not bipartite, a vertex is identified with a unique row-column pair. A directed graph $G_D = (V, E)$ with vertex set V and edge set E can be associated with an $n \times n$ sparse matrix $\mathbf{A}$. Here, $|V| = n$, and for each $a_{ij} \neq 0$ where $i \neq j$, we have a directed edge from v_i to v_j.

We also associate graphs with matrices. For a given $m \times n$ matrix $\mathbf{A}$, we can associate a bipartite graph $G = (V_R \cup V_C, E)$ so that the zero-nonzero pattern of $\mathbf{A}$ is equivalent to $\mathbf{A}_G$. If the matrix is square and has a symmetric zero-nonzero pattern, we can similarly associate an undirected graph by ignoring the diagonal entries. Finally, if the matrix $\mathbf{A}$ is square and has an unsymmetric zero-nonzero pattern, we can associate a directed graph $G_D = (V, E)$ where $(v_i, v_j) \in E$ iff $a_{ij} \neq 0$.

An $n \times n$ matrix $\mathbf{A} \neq 0$ is said to have *support* if there is a perfect matching in the associated bipartite graph. An $n \times n$ matrix $\mathbf{A}$ is said to have *total support* if each edge in its bipartite graph can be put into a perfect matching. A square sparse matrix is called *irreducible* if its directed graph is strongly connected. A square sparse matrix $\mathbf{A}$ is called *fully indecomposable* if for a permutation matrix $\mathbf{Q}$, the matrix $\mathbf{B} = \mathbf{AQ}$ has a zero free diagonal and the directed graph associated with $\mathbf{B}$ is irreducible. Fully indecomposable matrices have total support; but a matrix with total support could be a block diagonal matrix, where each block is fully indecomposable. For more formal definitions of support, total support, and the fully indecomposability, see for example the book by Brualdi and Ryser [34, Ch. 3 and Ch. 4].

Let $\mathbf{A} = [a_{ij}] \in \mathbb{R}^{n \times n}$. The matrix $\mathbf{A}$ is said to be doubly stochastic if $a_{ij} \geq 0$ for all i, j and $\mathbf{A}e = \mathbf{A}^T e = e$, where e is the vector of all ones. By Birkhoff's Theorem [25], there exist $\alpha_1, \alpha_2, \ldots, \alpha_k \in (0, 1)$ with $\sum_{i=1}^{k} \alpha_i = 1$ and permutation matrices $\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_k$ such that:

$$\mathbf{A} = \alpha_1 \mathbf{P}_1 + \alpha_2 \mathbf{P}_2 + \cdots + \alpha_k \mathbf{P}_k \,. \tag{1.1}$$

This representation is also called Birkhoff-von Neumann (BvN) decomposition. We refer to the scalars $\alpha_1, \alpha_2, \ldots, \alpha_k$ as the coefficients of the decomposition.

Doubly stochastic matrices and their associated BvN decompositions have been used in several operations research problems and applications. Classical examples are concerned with allocating communication resources, where an input traffic is routed to an output traffic in stages [43]. Each routing stage is a (sub-)permutation matrix and is used for handling a disjoint set of communications. The number of stages correspond to the number of (sub-)permutation matrices. A recent variant of this allocation problem

arises as a routing problem in data centers [146, 163]. Heuristics for the BvN decomposition have been used in designing business strategies for e-commerce retailers [149]. A display matrix (which shows what needs to be displayed in total) is constructed, which happens to be doubly stochastic. Then a BvN decomposition of this matrix is obtained in which each permutation matrix used is a display shown to a shopper. A small number of permutation matrices is preferred; furthermore, only slight changes between consecutive permutation matrices is desired (we do not deal with this last issue in this document).

The BvN decomposition of a doubly stochastic matrix as a convex combination of permutation matrices is not unique in general. The Marcus-Ree Theorem [170] states that there is always a decomposition with $k \leq n^2 - 2n + 2$ permutation matrices, for dense $n \times n$ matrices; Brualdi and Gibson [33] and Brualdi [32] show that for a fully indecomposable $n \times n$ sparse matrix with τ nonzeros one has the equivalent expression $k \leq \tau - 2n + 2$. In Chapter 5, we investigate the problem of finding the minimum number k of permutation matrices in the representation (1.1). More formally define the MinBvNDec problem defined as follows.

Problem 5 (MinBvNDec: A BvN decomposition with the minimum number of permutation matrices)**.** *Given a doubly stochastic matrix* $\mathbf{A}$*, find a BvN decomposition* (1.1) *of* $\mathbf{A}$ *with the minimum number* k *of permutation matrices.*

Brualdi [32, p.197] investigates the same problem and concludes that this is a difficult problem. We continue along this line and show that the MinBvNDec problem is NP-hard. We also propose a heuristic for obtaining a BvN decomposition with a small number of permutation matrices. We investigate some of the properties of the heuristic theoretically and experimentally. In this chapter, we also give a generalization of the BvN decomposition for real matrices with possibly negative entries and use this generalization for designing preconditioners for iterative linear system solvers.

1.5 Sparse tensors

Tensors are multidimensional arrays, generalizing matrices to higher orders. We use calligraphic font to refer to tensors, e.g., $\mathcal{X}$. Let $\mathcal{X}$ be a d-dimensional tensor whose size is $n_1 \times \cdots \times n_d$. As in matrices, an element of a tensor is denoted by a lowercase letter and subscripts corresponding to the indices of the element, e.g., $x_{i_1,\ldots,i_d}$, where $i_j \in \{1, \ldots, n_j\}$. A marginal of a tensor is a $(d-1)$-dimensional slice of a d-dimensional tensor, obtained by fixing one of its indices. A d-dimensional tensor where the entries in each of its marginals sum to one is called d-stochastic. In a d-stochastic tensor, all dimensions

necessarily have the same size n. A d-stochastic tensor where each marginal contains exactly one nonzero entry (equal to one) is called a permutation tensor.

A d-partite, d-uniform hypergraph $H = (V_1 \cup \cdots \cup V_d, E)$ corresponds naturally to a d-dimensional tensor $\mathcal{X}$, by associating each vertex class with a dimension of $\mathcal{X}$. Let $|V_i| = n_i$. Let $\mathcal{X} \in \{0,1\}^{n_1 \times \cdots \times n_d}$ be a tensor with nonzero elements $x_{v_1,\ldots,v_d}$ corresponding to the edges $(v_1, \ldots, v_d)$ of H. In this case, $\mathcal{X}$ is called the adjacency tensor of H.

Tensors are used in modeling multifactor or multirelational datasets, such as product reviews at an online store [21], natural language processing [123], multi-sensor data in signal processing [49], among many others [142]. Tensor decomposition algorithms are used as an important tool to understand the tensors and glean hidden or latent information. One of the most common tensor decomposition approaches is the Candecomp/Parafac (CP) formulation, which approximates a given tensor as a sum of rank-one tensors. More precisely, for a given positive integer R, called the decomposition rank, the CP decomposition approximates a d-dimensional tensor by a sum of R outer products of column vectors (one for each dimension). In particular, for a three dimensional $I \times J \times K$ tensor $\mathcal{X}$, CP asks for a rank R approximation $\mathcal{X} \approx \sum_{r=1}^{R} a_r \circ b_r \circ c_r$ with vectors a_r, b_r and c_r of size, respectively, I, J and K. Here, $\circ$ denotes the outer product of the vectors, and hence $x_{i,j,k} \approx \sum_{r=1}^{R} a_r(i) \cdot b_r(j) \cdot c_r(k)$. By aggregating these column vectors, one forms the matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ of size, respectively, $I \times R$, $J \times R$ and $K \times R$. The matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are called the factor matrices.

The computational core of the most common CP decomposition methods is an operation called the matricized tensor-times-Khatri-Rao product (MTTKRP). For the same tensor $\mathcal{X}$, this core operation can be described succinctly as

$$\begin{aligned} &\textbf{foreach } x_{i,j,k} \in \mathcal{X} \textbf{ do} \\ &\quad \tilde{\mathbf{A}}(i,:) \leftarrow \tilde{\mathbf{A}}(i,:) + x_{i,j,k}[\mathbf{B}(j,:) * \mathbf{C}(k,:)] \end{aligned} \tag{1.2}$$

where $\mathbf{B}$ and $\mathbf{C}$ are input matrices of sizes $J \times R$ and $K \times R$, respectively; $\mathbf{M}_A$ is the output matrix of size $I \times R$ which is initialized to zero before the loop. For each tensor nonzero $x_{i,j,k}$, the jth row of the matrix $\mathbf{B}$ is multiplied element-wise with the kth row of the matrix $\mathbf{C}$, the result is scaled with the tensor entry $x_{i,j,k}$ and added to the ith row of $\mathbf{M}_A$. In the well known CP decomposition algorithm based on alternative least squares (CP-ALS), the two factor matrices $\mathbf{B}$ and $\mathbf{C}$ are initialized at the beginning, then the factor matrix $\mathbf{A}$ is computed by multiplying $\mathbf{M}_A$ with the (pseudo-)inverse of the Hadamard product of $\mathbf{B}^T\mathbf{B}$ and $\mathbf{C}^T\mathbf{C}$. Since $\mathbf{B}^T\mathbf{B}$ and $\mathbf{C}^T\mathbf{C}$ are of size $R \times R$, their multiplication and inversion do not pose any computational challenges. Afterwards, the same steps are performed to compute a new $\mathbf{B}$, and then a new $\mathbf{C}$, and this process of computing the new factors one

at a time by assuming others fixed is repeated iteratively. In Chapter 3, we investigate how to efficiently parallelize the MTTKRP operations with the CP-ALS algorithm in distributed memory systems. In particular we investigate the following problem.

Problem 6 (Parallelizing CP decomposition)**.** *Given a sparse tensor $\mathcal{X}$, the rank R of the required decomposition, the number k of processors, partition $\mathcal{X}$ among the processors so as to achieve load balance and reduce the communication cost in a parallel implementation of* MTTKRP.

More precisely, in Chapter 3 we describe parallel implementations of CP-ALS and how the parallelization problem can be cast as a partitioning problem in hypergraphs. In this document, we do not develop heuristics for the hypergraph partitioning problem—we describe hypergraphs modeling the computations and make use of the well-established tools for partitioning these hypergraphs. The MTTKRP operation also arises in the gradient descent based methods [2] to compute CP decomposition. Furthermore, it has been implemented as a stand-alone routine [17] to enable algorithm development for variants of the core methods [158]. That is why it has been investigated for efficient execution in different computational frameworks such as Matlab [11, 18], MapReduce [123], shared memory [206], and distributed memory [47].

Consider again the MTTKRP operation and the execution of the operations as shown in the body of the displayed for-loop (1.2). As seen in this loop, the rows of the factor matrices and the output matrix are accessed many times according to the sparsity pattern of the tensor $\mathcal{X}$. In a cache-based system, the computations will be much more efficient if we take advantage of spatial and temporal locality. If we can reorder the nonzeros such that the multiplications involving a given row (of $\mathbf{M}_A$ and $\mathbf{B}$ and $\mathbf{C}$) are one after another, this would be achieved. We need this to hold when executing the MTTKRPs for computing the new factor matrices $\mathbf{B}$ and $\mathbf{C}$ as well. The issue is more easily understood in terms of the sparse matrix–vector multiplication operation $y \leftarrow \mathbf{A}x$. If we have two nonzeros in the same row a_{ij} and a_{ik}, we would prefer an ordering that maps j and k next to each other, and perform the associated scalar multiplications $a_{ij'}x_{j'}$ and $a_{ik'}x_{k'}$ one after other, where j' is the new index id for j and k' is the new index id for k. Furthermore, we would like the same sort of locality for all indices and for the multiplication $x \leftarrow \mathbf{A}^T y$ as well. We identify two related problems here. One of them is the choice of data structure for taking advantage of shared indices, and the other is to reorder the indices so that nonzeros sharing an index in a dimension have close-by indices in the other dimensions. In this document, we do not propose a data structure; our aim is to reorder the indices such that the locality is improved for three common data structures used in the current state of the art. We summarize those

three common data structures in Chapter 8, and investigate the following ordering problem. Since the overall problem is based on the data structure choice, we state the problem in a general language.

Problem 7 (Tensor ordering)**.** *Given a sparse tensor $\mathcal{X}$, find an ordering of the indices in each dimension so that the nonzeros of $\mathcal{X}$ have indices close to each other.*

If we look at the same problem for matrices, the problem can be cast as ordering the rows and the columns of a given matrix so that the nonzeros are clustered around the diagonal. This is also what we search in the tensor case, where different data structures can benefit from the same ordering, thanks to different effects. In Chapter 8, we investigate this problem algorithmically and develop fast heuristics. While the problem in the case of sparse matrices has been investigated with different angles, to the best of our knowledge, ours is the first study proposing general ordering tools for tensors—which are analogues of the well-known and widely used sparse matrix ordering heuristics such as Cuthill-McKee [55] and reverse Cuthill-McKee [95].

Part I

Partitioning

Chapter 2

Multilevel algorithms for acyclic partitioning of directed acyclic graphs

In this chapter, we address the directed acyclic graph partitioning (DAGP) problem. Most of the terms and definitions are standard, which we summarized in Section 1.1. The formal problem definition is repeated below for convenience.

DAG partitioning problem. Given a DAG $G = (V, E)$, an imbalance parameter ε, find an acyclic k-way partition $\{V_1, \ldots, V_k\}$ of V such that the balance constraints

$$w(V_i) \leq (1 + \varepsilon)\frac{w(V)}{k}$$

are satisfied for $1 \leq i \leq k$, and the edge cut is minimized.

We adopt the multilevel approach [35, 113] with the coarsening, initial partitioning, and refinement phases for acyclic partitioning of DAGs. We propose heuristics for these three phases (Sections 2.2.1, 2.2.2 and 2.2.3, respectively) which guarantee acyclicity of the partitions at all phases and maintain a DAG at every level. We strove to have fast heuristics at the core. With these characterizations, the coarsening phase requires new algorithmic/theoretical reasoning, while the initial partitioning and refinement heuristics are direct adaptations of the standard methods used in undirected graph partitioning, with some differences worth mentioning. We discuss only the bisection case, as we were able to improve the direct k-way algorithms we proposed before [114] by using the bisection heuristics recursively.

The acyclicity constraint on the partitions forbids the use of the state of the art undirected graph partitioning tools. This has been recognized

before, and those tools were put aside [114, 175]. While this is true, one can still try to make use of the existing undirected graph partitioning tools [113, 130, 186, 199], as they have been very well engineered. Let us assume that we have partitioned a DAG with an undirected graph partitioning tool into two parts by ignoring the directions. It is easy to detect if the partition is cyclic since all the edges need to go from the first part to the second one. If a partition is cyclic, we can easily fix it as follows. Let v be a vertex in the second part; we can move all vertices to which there is a path from v into the second part. This procedure breaks any cycle containing v, and, hence, the partition becomes acyclic. However, the edge cut may increase, and the partitions can be unbalanced. To solve the balance problem and reduce the cut, we can apply move-based refinement algorithms. After this step, the final partition meets the acyclicity and balance conditions. Depending on the structure of the input graph, it could also be a good initial partition for reducing the edge cut. Indeed, one of our most effective schemes uses an undirected graph partitioning algorithm to create a (potentially cyclic) partition, fixes the cycles in the partition, and refines the resulting acyclic partition with a novel heuristic to obtain an initial partition. We then integrate this partition within the proposed coarsening approaches to refine it at different granularities.

2.1 Related work

Fauzia et al. [87] propose a heuristic for the acyclic partitioning problem to optimize data locality when analyzing DAGs. To create partitions, the heuristic categorizes a vertex as ready to be assigned to a partition when all of the vertices it depends on have already been assigned. Vertices are assigned to the current partition set until the maximum number of vertices that would be "active" during the computation of the part reaches a specified limit, which is the cache size in their application. This implies that a part size is not limited by the sum of the total vertex weights in that part but is a complex function that depends on an external schedule (order) of the vertices. This differs from our problem as we limit the size of each part by the total sum of the weights of the vertices on that part.

Kernighan [137] proposes an algorithm to find a minimum edge-cut partition of the vertices of a graph into subsets of size greater than a lower bound and inferior to an upper bound. The partition needs to use a fixed vertex sequence that cannot be changed. Indeed, Kernighan's algorithm takes a topological order of the vertices of the graph as an input and partitions the vertices such that all vertices in a subset constitute a continuous block in the given topological order. This procedure is optimal for a given, fixed topological order and has a run time proportional to the number of edges in the graph, if the part weights are taken as constant. We used a

modified version of this algorithm as a heuristic in the earlier version of our work [114].

Cong et al. [54] describe two approaches for obtaining acyclic partitions of directed Boolean networks, modeling circuits. The first one is a single-level Fiduccia-Mattheyses (FM)-based approach. In this approach, Cong et al. generate an initial acyclic partition by splitting the list of the vertices (in a topological order) from left to right into k parts such that the weight of each part does not violate the bound. The quality of the results is then improved with a k-way variant of the FM heuristic [88] taking the acyclicity constraint into account. Our previous work [114] employs a similar refinement heuristic. The second approach of Cong et al. is a two-level heuristic; the initial graph is first clustered with a special decomposition, and then it is partitioned using the first heuristic.

In a recent paper [175], Moreira et al. focus on an imaging and computer vision application on embedded systems and discuss acyclic partitioning heuristics. They propose a single level approach in which an initial acyclic partitioning is obtained using a topological order. To refine the partitioning, they proposed four local search heuristics which respect the balance constraint and maintain the acyclicity of the partition. Three heuristics pick a vertex and move it to an eligible part if and only if the move improves the cut. These three heuristics differ in choosing the set of eligible parts for each vertex; some are very restrictive, and some allow arbitrary target parts as long as acyclicity is maintained. The fourth heuristic tentatively realizes the moves that increase the cut in order to escape from a possible local minima. It has been reported that this heuristic delivers better results than the others. In a follow-up paper, Moreira et al. [176] discuss a multilevel graph partitioner and an evolutionary algorithm based on this multilevel scheme. Their multilevel scheme starts with a given acyclic partition. Then, the coarsening phase contracts edges that are in the same part until there is no edge to contract. Here, matching-based heuristics from undirected graph partitioning tools are used without taking the directions of the edges into account. Therefore, the coarsening phase can create cycles in the graph; however the induced partitions are never cyclic. Then, an initial partition is obtained, which is refined during the uncoarsening phase with move-based heuristics. In order to guarantee acyclic partitions, the vertices that lie in cycles are not moved. In a systematic evaluation of the proposed methods, Moreira et al. note that there are many local minima and suggest using relaxed constraints in the multilevel setting. The proposed methods have high run time, as the evolutionary method of Moreira et al. is not concerned with this issue. Improvements with respect to the earlier work [175] are reported.

We propose methods to use an undirected graph partitioner to guide the multilevel partitioner. We focus on partitioning the graph in two parts, since one can handle the general case with a recursive bisection scheme. We also propose new coarsening, initial partitioning, and refinement methods

specifically designed for the 2-partitioning problem. Our multilevel scheme maintains acyclic partitions and graphs through all the levels.

2.2 Directed multilevel graph partitioning

Here, we summarize the proposed methods for the three phases of the multilevel scheme.

2.2.1 Coarsening

In this phase, we obtain smaller DAGs by coalescing the vertices, level by level. This phase continues until the number of vertices becomes smaller than a specified bound or the reduction on the number of vertices from one level to the next one is lower than a threshold. At each level ℓ, we start with a finer acyclic graph G_ℓ, compute a valid clustering $\mathcal{C}_\ell$ ensuring the acyclicity, and obtain a coarser acyclic graph $G_{\ell+1}$. While our previous work [114] discussed matching based algorithms for coarsening, we present agglomerative clustering based variants here. The new variants supersede the matching based ones. Unlike the standard undirected graph case, in DAG partitioning, not all vertices can be safely combined. Consider a DAG with three vertices a, b, c and three edges (a, b), (b, c), (a, c). Here, the vertices a and c cannot be combined, since that would create a cycle. We say that a set of vertices is contractible (all its vertices are matchable), if unifying them does not create a cycle. We now present a general theory about finding clusters without forming cycles, after giving some definitions.

Definition 2.1 (**Clustering).** *A clustering of a DAG is a set of disjoint subsets of vertices. Note that we do not make any assumptions on whether the subsets are connected or not.*

Definition 2.2 (**Coarse graph).** *Given a DAG G and a clustering C of G, we let $G_{|C}$ denote the coarse graph created by contracting all sets of vertices of C.*

The vertices of the coarse graph are the clusters in C. If $(u, v) \in G$ for two vertices u and v that are located in different clusters of C then $G_{|C}$ has an (directed) edge from the vertex corresponding to u's cluster, to the vertex corresponding to v's cluster.

Definition 2.3 (**Feasible clustering).** *A feasible clustering C of a DAG G is a clustering such that $G_{|C}$ is acyclic.*

Theorem 2.1. *Let $G = (V, E)$ be a DAG. For $u, v \in V$ and $(u, v) \in E$, the coarse graph $G_{|\{(u,v)\}}$ is acyclic if and only if every path from u to v in G contains the edge (u, v).*

Theorem 2.1, whose proof is in the journal version of this chapter [115], can be extended to a set of vertices by noting that this time all paths connecting two vertices of the set should contain only the vertices of the set. The theorem (nor its extension) does not imply an efficient algorithm, as it requires at least one transitive reduction. Furthermore, it does not describe a condition about two clusters forming a cycle, even if both are individually contractible. In order to address both of these issues, we put a constraint on the vertices that can form a cluster, based on the following definition.

Definition 2.4 (**Top level value**). *For a DAG $G = (V, E)$, the top level value of a vertex $u \in V$ is the length of the longest path from a source of G to that vertex. The top level values of all vertices can be computed in a single traversal of the graph with a complexity $O(|V| + |E|)$. We use* $\texttt{top}[u]$ *to denote the top level of the vertex u.*

By restricting the set of edges considered in the clustering to the edges $(u, v) \in E$ such that $\texttt{top}[u]+1 = \texttt{top}[v]$, we ensure that no cycles are formed by contracting a unique cluster (the condition identified in Theorem 2.1 is satisfied). Let C be a clustering of the vertices. Every edge in a cluster of C being contractible is a necessary condition for C to be feasible, but not a sufficient one. More restrictions on the edges of vertices inside the clusters should be found to ensure that C is feasible. We propose three coarsening heuristics based on clustering sets of more than two vertices, whose pair-wise top level differences are always zero or one.

`CoTop`: Acyclic clustering with forbidden edges

To have an efficient clustering heuristic, we restrict ourselves to static information computable in linear time. As stated in the introduction of this section, we rely on the top level difference of one (or less) for all vertices in the same cluster, and an additional condition to ensure that there will be no cycles when a number of clusters are contracted simultaneously. In Theorem 2.2 below, we give two sufficient conditions for a clustering to be feasible (that is, the graphs at all levels are DAGs)—the proof is in the associated journal paper [115].

Theorem 2.2 (Correctness of the proposed clustering). *Let $G = (V, E)$ be a DAG and $C = \{C_1, \ldots, C_k\}$ be a clustering. If C is such that:*

- *for any cluster C_i, for all $u, v \in C_i$, $|\texttt{top}[u] - \texttt{top}[v]| \leq 1$,*
- *for two different clusters C_i and C_j and for all $u \in C_i$ and $v \in C_j$ either $(u, v) \notin E$, or $\texttt{top}[u] \neq \texttt{top}[v] - 1$,*

then, the coarse graph $G_{|C}$ is acyclic.

We design a heuristic based on Theorem 2.2. This heuristic visits all vertices, which are singleton at the beginning, in an order, and adds the visited vertex to a cluster, if the criteria mentioned in Theorem 2.2 are met; if not, the vertex stays as a singleton. Among those clusters satisfying the mentioned criteria, the one which saves the largest edge weight (incident on the vertices of the cluster and the vertex being visited) is chosen as the target cluster. We discuss a set of bookkeeping operations in order to implement this heuristic in linear, $O(|V| + |E|)$, time [115, Algorithm 1].

`CoCyc`: Acyclic clustering with cycle detection

We now propose a less restrictive clustering algorithm which also maintains the acyclicity. We again rely on the top level difference of one (or less) for all vertices in the same cluster. Knowing this invariant, when a new vertex is added to a cluster, a cycle-detection algorithm checks that no cycles are formed when all the clusters are contracted simultaneously. This algorithm does not traverse the entire graph by also using the fact that the top level difference within a cluster is at most one. Nonetheless, detecting such cycles could require a full traversal of the graph.

From Theorem 2.2, we know if adding a vertex to a cluster whose vertices' top level values are t and $t+1$ creates a cycle in the contracted graph, then this cycle goes through the vertices with top level values t or $t+1$. Thus, when considering the addition of a vertex u to a cluster C containing v, we check potential cycle formations by traversing the graph starting from u in a breadth-first search (BFS) manner. Let t denote the minimum top level in C. At a vertex w, we normally add a successor y of w in a queue (as in BFS), if $|\texttt{top}(y) - t| \leq 1$; if w is in the same cluster as one of its predecessors x, we also add x to the queue if $|\texttt{top}(x) - t| \leq 1$. We detect a cycle if at some point in the traversal, a vertex from cluster C is reached, if not the cluster can be extended by adding u. In the worst-case, the cycle detection algorithm completes a full graph traversal but in practice, it stops quickly and does not introduce a significant overhead. The details of this clustering scheme, whose worst case run time complexity is $O(|V|(|V| + |E|))$, can be found in the original paper [115, Algorithm 2].

`CoHyb`: Hybrid acyclic clustering

The cycle detection based algorithm can suffer from quadratic run time for vertices with large in-degrees or out-degrees. To avoid this, we design a hybrid acyclic clustering which uses the relaxed clustering strategy with cycle detection `CoCyc` by default and switches to the more restrictive clustering strategy `CoTop` for *large degree* vertices. We define a limit on the degree of a vertex (typically $\sqrt{|V|}/10$) for calling it *large degree*. When considering an edge (u, v) where $\texttt{top}[u] + 1 = \texttt{top}[v]$, if the degrees of u and v do not

exceed the limit, we use the cycle detection algorithm to determine if we can contract the edge. Otherwise, if the outdegree of u or the indegree of v is too large, the edge will be contracted if it is allowed. The complexity of this algorithm is in between those `CoTop` and `CoCyc`, and it will likely avoid the quadratic behavior in practice.

2.2.2 Initial partitioning

After the coarsening phase, we compute an initial acyclic partitioning of the coarsest graph. We present two heuristics. One of them is akin to the greedy graph growing method used in the standard graph/hypergraph partitioning methods. The second one uses an undirected partitioning and then fixes the acyclicity of the partitions. Throughout this section, we use (V_0, V_1) to denote the bisection of the vertices of the coarsest graph, and ensure that there is no edge from the vertices in V_1 to those in V_0.

Greedy directed graph growing

One approach to compute a bisection of a directed graph is to design a greedy algorithm that moves vertices from one part to another using local information. We start with all vertices in V_1 and replace vertices towards V_0 by using heaps. At any time, the vertices that can be moved to V_0 are in the heap. These vertices are those whose in-neighbors are all in V_0. Initially only the sources are in the heap, and when all the in-neighbors of a vertex v are moved to C_0, v is inserted into the heap. We separate this process into two phases. In the first phase, the key-values of the vertices in the heap are set to the weighted sum of their incoming edges, and the ties are broken in favor of the vertex closer to the first vertex moved. The first phase continues until the first part has more than 0.9 of the maximum allowed weight (modulo the maximum weight of a vertex). In the second phase, the actual gain of a vertex is used. This gain is equal to the sum of the weights of the incoming edges minus the sum of the weights of the outgoing edges. In this phase, the ties are broken in favor of the heavier vertices. The second phase stops as soon as the required balance is obtained. The reason that we separated this heuristic into two phases is that at the beginning, the gains are of no importance, and the more vertices become movable the more flexibility the heuristic has. Yet, towards the end, parts are fairly balanced, and using actual gains should help keeping the cut small.

Since the order of the parts is important, we also reverse the roles of the parts, and the directions of the edges. That is, we put all vertices in V_0, and move the vertices one by one to V_1, when all out-neighbors of a vertex have been moved to V_1. The proposed greedy directed graph growing heuristic returns the best of the these two alternatives.

Undirected bisection and fixing acyclicity

In this heuristic, we partition the coarsest graph as if it were undirected, and then move the vertices from one part to another in case the partition was not acyclic. Let (P_0, P_1) denote the (not necessarily acyclic) bisection of the coarsest graph treated as if it were undirected.

The proposed approach designates arbitrarily P_0 as V_0 and P_1 as V_1. One way to fix the cycle is to move all ancestors of the vertices in V_0 to V_0, thereby guaranteeing that there is no edge from vertices in V_1 to vertices in V_0, making the bisection (V_0, V_1) acyclic. We do these moves in a reverse topological order. Another way to fix the acyclicity is to move all descendants of the vertices in V_1 to V_1, again guaranteeing an acyclic partition. We do these moves in a topological order. We then fix the possible unbalance with a refinement algorithm.

Note that we can also initially designate P_1 as V_0 and P_0 as V_1, and can again fix a potential cycle in two different ways. We try all four of these choices, and return the best partition (essentially returning the best of the four choices to fix the acyclicity of (P_0, P_1)).

2.2.3 Refinement

This phase projects the partition obtained for a coarse graph to the next, finer one and refines the partition by vertex moves. As in the standard refinement methods, the proposed heuristic is applied in a number of passes. Within a pass, we repeatedly select the vertex with the maximum move gain among those that can be moved. We tentatively realize this move if the move maintains or improves the balance. Then, the most profitable prefix of vertex moves are realized at the end of the pass. As usual, we allow the vertices move only once in a pass. We use heaps with the gain of moves as the key value, where we keep only movable vertices. We call a vertex *movable*, if moving it to the other part does not create a cyclic partition. We use the notation (V_0, V_1) to designate the acyclic bisection with no edge from vertices in V_1 to vertices in V_0. This means that for a vertex to move from part V_0 to part V_1, one of the two conditions should be met (i) either all its out-neighbors should be in V_1; (ii) or the vertex has no out-neighbors at all. Similarly, for a vertex to move from part V_1 to part V_0, one of the two conditions should be met (i) either all its in-neighbors should be in V_0; (ii) or the vertex has no in-neighbors at all. This is an adaptation of the boundary Fiduccia-Mattheyses heuristic [88] to directed graphs. The notion of movability being more restrictive results in an important simplification. The gain of moving a vertex v from V_0 to V_1 is

$$\sum_{u \in \texttt{Succ}[v]} w(v,u) - \sum_{u \in \texttt{Pred}[v]} w(u,v) \ , \tag{2.1}$$

and the negative of this value when moving it from V_1 to V_0. This means that the gain of a vertex is static: once a vertex is inserted in the heap with the key value (2.1), its key-value is never updated. A move could render some vertices unmovable; if they were in the heap, then they should be deleted. Therefore, the heap data structure needs to support insert, delete, and extract max operations only.

2.2.4 Constrained coarsening and initial partitioning

There are a number of highly successful, undirected graph partitioning tools [130, 186, 199]. They are not immediately usable for our purposes, as the partitions can be cyclic. Fixing such partitions, by moving vertices to break the cyclic dependencies among the parts, can increase the edge cut dramatically (with respect to the undirected cut). Consider for example, the $n \times n$ grid graph, where the vertices are at integer positions for $i = 1, \ldots, n$ and $j = 1, \ldots, n$ and a vertex at (i, j) is connected to (i', j') when $|i - i'| = 1$ or $|j - j'| = 1$, but not both. There is an acyclic orientation of this graph, called spiral ordering, as described in Figure 2.1 for $n = 8$. This spiral ordering defines a total order. When the directions of the edges are ignored, we can have a bisection with perfect balance by cutting only $n = 8$ edges with a vertical line. This partition is cyclic; and it can be made acyclic by putting all vertices numbered greater than 32 to the second part. This partition, which puts the vertices 1–32 to the first part and the rest to the second part, is the unique acyclic bisection with perfect balance for the associated directed acyclic graph. The edge cut in the directed version is 35 as seen in the figure (gray edges). In general one has to cut $n^2 - 4n + 3$ edges for $n \geq 8$: the blue vertices in the border (excluding the corners) have one edge directed to a red vertex; the interior blue vertices have two such edges; finally, the blue vertex labeled $n^2/2$ has three such edges.

Let us investigate the quality of the partitions in practice. We used MeTiS [130] as the undirected graph partitioner on a dataset of 94 matrices (their details are in Section 2.3). The results are given in Figure 2.2. For this preliminary experiment, we partitioned the graphs into two with the maximum allowed load imbalance $\varepsilon = 3\%$. The output of MeTiS is acyclic for only two graphs, and the geometric mean of the normalized edge cut is 0.0012. Figure 2.2a shows the normalized edge cut and the load imbalance after fixing the cycles, while Figure 2.2b shows the two measurements after meeting the balance criteria. A normalized edge cut value is computed by normalizing the edge cut with respect to the number of edges.

In both figures, the horizontal lines mark the geometric mean of the normalized edge cuts, and the vertical lines mark the 3% imbalance ratio. In Figure 2.2a, there are 37 instances in which the load balance after fixing the cycles is feasible. The geometric mean of the normalized edge cuts in this sub-figure is 0.0045, while in the other sub-figure, it is 0.0049. Fixing

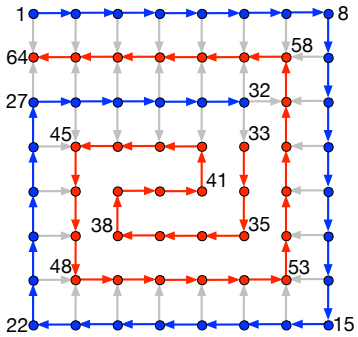
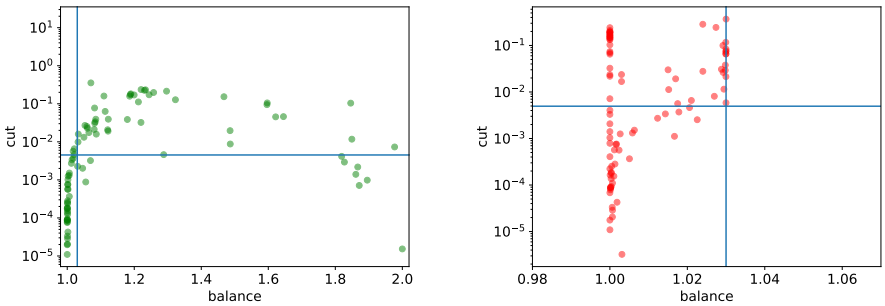


Figure 2.1: 8×8 grid graph whose vertices are ordered in a spiral way; a few of the vertices are labeled. All edges are oriented from a lower numbered vertex to a higher numbered one. There is a unique bipartition with 32 vertices in each side. The edges defining the total order are shown in red and blue, except the one from 32 to 33; the cut edges are shown in gray; other internal edges are not shown.



(a) Undirected partitioning and fixing cycles

(b) Undirected partitioning, fixing cycles, and balancing

Figure 2.2: Normalized edge cut (normalized with respect to the number of edges), the balance obtained after using an undirected graph partitioner and fixing the cycles (left), and after ensuring balance with refinement (right).

the cycles increases the edge cut with respect to an undirected partitioning, but not catastrophically (only by 0.0045/0.0012 = 3.75 times in these experiments), and achieving balance after this step increases the cut only a little (goes to 0.0049 from 0.0045). That is why we suggest using an undirected graph partitioner, fixing the cycles among the parts, and performing a refinement based method for load balancing as a good (initial) partitioner.

In order to refine the initial partition in a multilevel setting, we propose a scheme similar to the *iterated multilevel algorithm* used in the existing partitioners [40, 212]. In this scheme, first a partition P is obtained. Then, the coarsening phase starts to cluster vertices that were in the same part in P. After the coarsening, an initial partitioning is freely available by using the partition P on the coarsest graph. The refinement phase then can work as before. Moreira et al. [176] use this approach for the directed graph partitioning problem. To be more concrete, we first use an undirected graph partitioner, then fix the cycles as discussed in Section 2.2.2, and then refine this acyclic partition for balance with the proposed refinement heuristics in Section 2.2.3. We then use this acyclic partition for constrained coarsening and initial partitioning. We expect this scheme to be successful in graphs with many sources and targets where the sources and targets can lie in any of the parts while the overall partition is acyclic. On the other hand, if a graph is such that its balanced acyclic partitions need to put sources in one part and the targets in another part, then fixing acyclicity may result in moving many vertices. This in turn will harm the edge cut found by the undirected graph partitioner.

2.3 Experiments

The partitioning tool presented (`dagP`) is implemented in C/C++ programming languages. The experiments are conducted on a computer equipped with dual 2.1 GHz, Xeon E5-2683 processors and 512GB memory. The source code and more information is available at `http://tda.gatech.edu/software/dagP/`.

We have performed a large set of experiments on DAG instances coming from two sources. The first set of instances is from the Polyhedral Benchmark suite (PolyBench) [196]. The second set of instances is obtained from the matrices available in the SuiteSparse Matrix Collection (UFL) [59]. From this collection, we pick all the matrices satisfying the following properties: listed as binary, square, and has at least 100000 rows and at most 2^{26} nonzeros. There were a total of 95 matrices at the time of experimentation, where two matrices (ids 1514 and 2294) having the same pattern. We discard the duplicate and use the remaining 94 matrices for experiments. For each such matrix, we take the strict upper triangular part as the associated DAG instance, whenever this part has more nonzeros than the lower

Graph	*#vertex*	*#edge*	max. deg.	avg. deg.	*#source*	*#target*
2mm	36,500	62,200	40	1.704	2100	400
3mm	111,900	214,600	40	1.918	3900	400
adi	596,695	1,059,590	109,760	1.776	843	28
atax	241,730	385,960	230	1.597	48530	230
covariance	191,600	368,775	70	1.925	4775	1275
doitgen	123,400	237,000	150	1.921	3400	3000
durbin	126,246	250,993	252	1.988	250	249
fdtd-2d	256,479	436,580	60	1.702	3579	1199
gemm	1,026,800	1,684,200	70	1.640	14600	4200
gemver	159,480	259,440	120	1.627	15360	120
gesummv	376,000	500,500	500	1.331	125250	250
heat-3d	308,480	491,520	20	1.593	1280	512
jacobi-1d	239,202	398,000	100	1.664	402	398
jacobi-2d	157,808	282,240	20	1.789	1008	784
lu	344,520	676,240	79	1.963	6400	1
ludcmp	357,320	701,680	80	1.964	6480	1
mvt	200,800	320,000	200	1.594	40800	400
seidel-2d	261,520	490,960	60	1.877	1600	1
symm	254,020	440,400	120	1.734	5680	2400
syr2k	111,000	180,900	60	1.630	2100	900
syrk	594,480	975,240	81	1.640	8040	3240
trisolv	240,600	320,000	399	1.330	80600	1
trmm	294,570	571,200	80	1.939	6570	4800

Table 2.1: DAG instances from PolyBench [196].

triangular part; otherwise we take the strict lower triangular part. All edges have unit cost, and all vertices have unit weight.

Since the proposed heuristics have a randomized behavior (the traversals used in the coarsening and refinement heuristics are randomized), we run them 10 times for each DAG instance, and report the averages of these runs. We use performance profiles [69] to present the edge-cut results. A performance profile plot shows the probability that a specific method gives results within a factor θ of the best edge cut obtained by any of the methods compared in the plot. Hence, the higher and closer a plot to the y-axis, the better the method is.

We set the load imbalance parameter $\varepsilon = 0.03$ in the problem definition given at the beginning of the chapter for all experiments. The vertices are unit weighted, therefore, the imbalance is rarely an issue for a move-based partitioner.

Coarsening evaluation

We first evaluated the proposed coarsening heuristics (Section 5.1 of the associated paper [115]). The aim was to find an effective one to set as a default coarsening heuristic. Based on performance profiles on the edge cut, we concluded that in general, the coarsening heuristics `CoHyb` and `CoCyc` behave similarly and are more helpful than `CoTop` in reducing the edge cut. We also analyzed the contraction efficiency of the proposed coarsening heuristics. It is important that the coarsening phase does not stop too early and that the

coarsest graph is small enough to be partitioned efficiently. By investigating the maximum, the average, and the standard deviation of vertex and edge weight ratios at the coarsest graph, and the average, the minimum, and the maximum number of coarsening levels observed for the two datasets, we saw that `CoCyc` and `CoHyb` behave again similarly and provide slightly better results than `CoTop`. Based on all these observations, we set `CoHyb` as the default coarsening heuristic, as it performs better than `CoTop` in terms of final edge cut, and is guaranteed to be more efficient than `CoCyc` in terms of run time.

Constrained coarsening and initial partitioning

We now investigate the effect of using undirected graph partitioners for coarsening and initial partitioning as explained in Section 2.2.4. We compare three variants of the proposed multilevel scheme. All of them use the refinement described in Section 2.2.3 in the uncoarsening phase.

- `CoHyb`: this variant uses the hybrid coarsening heuristic described in Section 2.2.1 and the greedy directed graph growing heuristic described in Section 2.2.2 in the initial partitioning phase. This method does not use constrained coarsening.
- `CoHyb_C`: this variant uses an acyclic partition of the finest graph obtained as outlined in Section 2.2.2 to guide the hybrid coarsening heuristic described in Section 2.2.4, and uses the greedy directed graph growing heuristic in the initial partitioning phase.
- `CoHyb_CIP`: this variant uses the same constrained coarsening heuristic as the previous method, but inherits the fixed acyclic partition of the finest graph as the initial partitioning.

The comparison of these three variants are given in Figure 2.3 for the whole dataset. In this figure, we see that using the constrained coarsening is always helpful, as it clearly separates `CoHyb_C` and `CoHyb_CIP` from `CoHyb` after $\theta = 1.1$. Furthermore, applying the constrained initial partitioning (on top of the constrained coarsening) brings tangible improvements.

In the light of the experiments presented here, we suggest the variant `CoHyb_CIP` for general problem instances.

`CoHyb_CIP` with respect to a single level algorithm

We compare `CoHyb_CIP` (constrained coarsening and initial partitioning) with a single-level algorithm that uses an undirected graph partitioning, fixes the acyclicity, and refines the partitions. This last variant is denoted as `UndirFix`, and it is the algorithm described in Section 2.2.2. Both variants use the same initial partitioning approach, which utilizes MeTiS [130] as

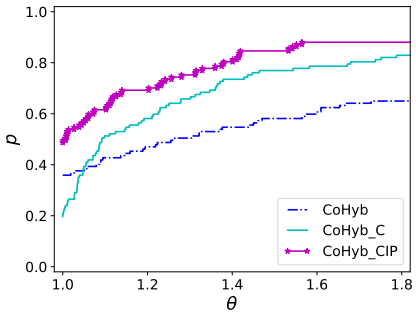


Figure 2.3: Performance profiles for the edge cut obtained by the proposed multilevel algorithm using the constrained coarsening and partitioning (`CoHyb_CIP`), using the constrained coarsening and the greedy directed graph growing (`CoHyb_C`), and `CoHyb` (no constraints).

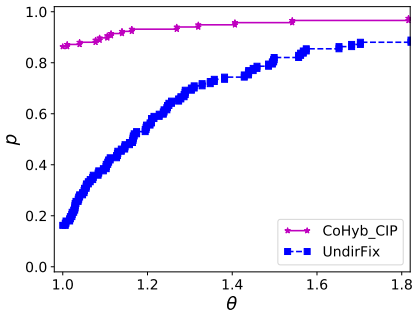


Figure 2.4: Performance profiles for the edge cut obtained by the proposed multilevel algorithm using the constrained coarsening and partitioning (`CoHyb_CIP`) and using the same approach without coarsening (`UndirFix`).

the undirected partitioner. The difference between `UndirFix` and `CoHyb_CIP` is the latter's ability to refine the initial partition at multiple levels. Figure 2.4 presents this comparison. The plots show that the multilevel scheme `CoHyb_CIP` outperforms the single level scheme `UndirFix` at all appropriate ranges of θ, attesting to the importance of the multilevel scheme.

Comparison with existing work

Here we compare our approach with the evolutionary graph partitioning approach developed by Moreira et al. [175], and briefly with our previous work [114].

Figure 2.5 shows how `CoHyb_CIP` and `CoTop` compare with the evolutionary approach in terms of the edge cut on the 23 graphs of the PolyBench dataset, for the number of partitions $k \in \{2, 4, 8, 16, 32\}$. We use the av-

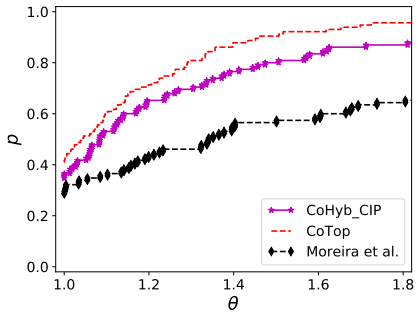


Figure 2.5: Performance profiles for the edge cut obtained by `CoHyb_CIP`, `CoTop`, and Moreira et al.'s approach on the PolyBench dataset with $k \in \{2, 4, 8, 16, 32\}$.

erage edge cut value of 10 runs for `CoTop` and `CoHyb_CIP` and the average values presented in [175] for the evolutionary algorithm. As seen in the figure, the `CoTop` variant of the proposed multilevel approach obtains the best results on this specific dataset (all variants of the proposed approach outperform the evolutionary approach). In more detailed comparisons [115, Appendix A], we see that the variants `CoHyb_CIP` and `CoTop` of the proposed algorithm obtain strictly better results than the evolutionary approach in 78 and 75 instances (out of 115), respectively, when the average edge cuts are compared.

In more detailed comparisons [115, Appendix A], we see that `CoHyb_CIP` obtains 26% less edge cut than the evolutionary approach on average (geometric mean) when the average cuts are compared: when the best cuts are compared, `CoHyb_CIP` obtains 48% less edge cut. Moreover, `CoTop` obtains 37% less edge cut than the evolutionary approach when the average cuts are compared; when the best cuts are compared, `CoTop` obtains 41% less cut. In some instances, we see large differences between the average and the best results of `CoTop` and `CoHyb_CIP`. Combined with the observation that `CoHyb_CIP` yields better results in general, this suggests that the neighborhood structure can be improved (see the notion of the strength of a neighborhood [183, Section 19.6]).

The proposed approach with all the reported variants takes about 30 minutes to complete the whole set of experiments for this dataset, whereas the evolutionary approach is much more compute-intensive, as it has to run the multilevel partitioning algorithm numerous times to create and update the population of partitions for the evolutionary algorithm. The multilevel approach of Moreira et al. [175] is more comparable in terms of characteristics with our multilevel scheme. When we compare `CoTop` with the results of the multilevel algorithm by Moreira et al., our approach provides results that are 37% better on average and `CoHyb_CIP` approach provides results that are

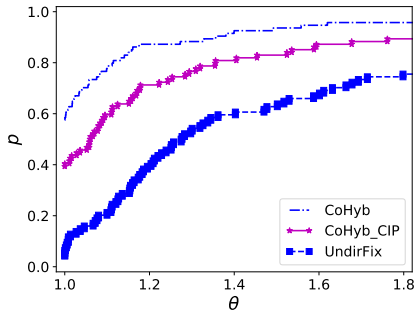


Figure 2.6: Performance profiles of `CoHyb`, `CoHyb_CIP` and `UndirFix` in terms of edge cut for single source, single target graph dataset. The average of 5 runs are reported for each approach.

26% better on average, highlighting the fact that keeping the acyclicity of the directed graph through the multilevel process is useful.

Finally, `CoTop` and `CoHyb_CIP` also outperform the previous version of our multilevel partitioner [114], which is based on a direct k-way partitioning scheme and matching heuristics for the coarsening phase, by 45% and 35% on average, respectively, on the same dataset.

Single commodity flow-like problem instances

In many of the instances of our dataset, graphs have many source and target vertices. We investigate how our algorithm performs on problems where all source vertices should be in a given part, and all target vertices should be in the other part, while also achieving balance. This is a problem close to the maximum flow problem, where we want to find the maximum flow (or minimum cut) from the sources to the targets with balance on part weights. Furthermore, addressing this problem also provides a setting for solving partitioning problems with fixed vertices.

We used the UFL dataset, where all isolated vertices are discarded, and a single source vertex S and a single target vertex T are added to all graphs. S is connected to all original source vertices, and all original target vertices are connected to T. The cost of the new edges is set to the number of edges. A feasible partition should avoid cutting these edges, and separate all sources from the targets.

The performance profiles of `CoHyb`, `CoHyb_CIP` and `UndirFix` are given in Figure 2.6 with the edge cut as the evaluation criterion. As seen in this figure, `CoHyb` is the best performing variant, and `UndirFix` is the worst performing variant. This is interesting as in the general setting, we saw a reverse relation. The variant `CoHyb_CIP` performs in the middle, as it combines the other two.

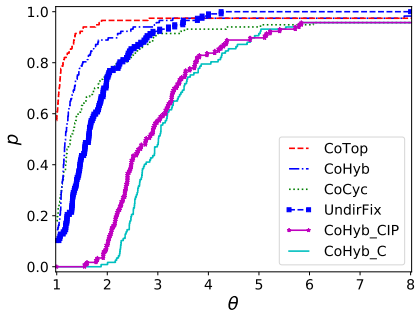


Figure 2.7: Run time performance profile of `CoCyc`, `CoHyb`, `CoTop`, `CoHyb_C`, `CoHyb_CIP` and `UndirFix` on the whole dataset. The values are the averages of 10 runs.

Run time performance

We give again a summary of the run time comparisons from the detailed version [115, Section 5.6]. Figure 2.7 shows the comparison of the five variants of the proposed multilevel scheme and the single level scheme on the whole dataset. Each algorithm is run 10 times on each graph. As expected, `CoTop` offers the best performance, and `CoHyb` offers a good trade-off between `CoTop` and `CoCyc`. An interesting remark is that these three algorithms have a better run time than the single level algorithm `UndirFix`. For example, on the average, `CoTop` is 1.44 times faster than `UndirFix`. This is mainly due to cost of fixing acyclicity. Undirected partitioning accounts for roughly 25% of the execution time of `UndirFix`, and fixing the acyclicity constitutes the remaining 75%. Finally, the variants of the multilevel algorithm using constrained coarsening heuristics provide satisfying run time performance with respect to the others.

2.4 Summary, further notes and references

We proposed a multilevel approach for acyclic partitioning of directed acyclic graphs. This problem is close to the standard graph partitioning in that the aim is to partition the vertices into a number of parts while minimizing the edge cut and meeting a balance criterion on the part weights. Unlike the standard graph partitioning problem, the directions of the edges are important and the resulting partitions should have acyclic dependencies.

We proposed coarsening, initial partitioning, and refinement heuristics for the target problem. The proposed heuristics take the directions of the edges into account and maintain the acyclicity throughout the three phases of the multilevel scheme. We also proposed efficient and effective approaches to use the standard undirected graph partitioning tools in the multilevel

scheme for coarsening and initial partitioning. We performed a large set of experiments on a dataset with graphs having different characteristics and evaluated different combinations of the proposed heuristics. Our experiments suggested (i) the use of constrained coarsening and initial partitioning, where the main coarsening heuristic is a hybrid one which avoids the cycles, and in case it does not, performs a fast cycle detection (`CoHyb_CIP`) for the general case; (ii) a pure multilevel scheme without constrained coarsening, using the hybrid coarsening heuristic (`CoHyb`) for the cases where a number of sources need to be separated from a number of targets; (iii) a pure multilevel scheme without constrained coarsening, using the fast coarsening algorithm (`CoTop`) for the cases where the degrees of the vertices are small. All three approaches are shown to be more effective and efficient than the current state of the art.

Özkaya et al. [181] make use of the proposed partitioner for scheduling directed acyclic task graphs for homogeneous computing systems. Here, the partitioner is used to obtain coarse grained tasks to enhance data locality. Lepère and Trystram [151] show that acyclic clustering is very effective in scheduling DAGs with unit task weights and large, uniform edge costs. In particular they show that there exists an acyclic clustering whose makespan is at most twice the best clustering.

Recall that a directed edge (u, v) is called redundant, if there is a path $u \rightsquigarrow v$ in the graph. Consider a redundant edge (u, v) and a path $u \rightsquigarrow v$. Let us discard the edge (u, v) and add its cost to all edges in the identified $u \rightsquigarrow v$ path to obtain a reduced graph. In an acyclic bisection of this reduced graph, if the vertices u and v are in different parts, then the path $u \rightsquigarrow v$ crosses the cut only once (otherwise the partition is cyclic), and the edge cut in the original graph equals the edge cut in the reduced graph. If u and v are in the same part, then all paths $u \rightsquigarrow v$ are confined to the same part. Therefore, neither the edge (u, v) of the original graph nor any path $u \rightsquigarrow v$ of the reduced graph contributes to the respective cuts. Hence the edge cut in the original graph is equal to the edge cut in the reduced graph. Such reductions could potentially help reduce the run time and improve the quality of the partitions in our multilevel setting. The transitive reduction is a costly operation, and hence we do not hope to apply all reductions in the proposed approach; a subset of those should be found and applied. The effects of such reductions should be more tangible in the evolutionary algorithm [175, 176].

Chapter 3

Parallel Candecomp/Parafac decomposition of sparse tensors in distributed memory

In this chapter, we address the problem of parallelizing the computational core of the Candecomp/Parafac decomposition of sparse tensors. Tensor notation is a little less common, and the reader is invited to revise the notation and the basic definitions in Section 1.5. More related definitions are given in this chapter. Below we restate the problem for convenience.

Parallelizing CP decomposition. Given a sparse tensor $\mathcal{X}$, the rank R of the required decomposition, the number k of processors, partition $\mathcal{X}$ among the processors so as to achieve load balance and reduce the communication cost in a parallel implementation of Mttkrp.

We investigate an efficient parallelization of the Mttkrp operation in distributed memory environments for sparse tensors in the context of the CP decomposition method CP-ALS. For this purpose, we formulate two task definitions, a coarse-grain and a fine-grain one. These definitions are given by applying the owner-computes rule to a coarse-grain and a fine-grain partition of the tensor nonzeros. We define the coarse-grain partition of a tensor as a partition of one of its dimensions. In matrix terms, a coarse-grain partition corresponds to a row-wise or a column-wise partition. We define the fine-grain partition of a tensor as a partition of its nonzeros. This has the same significance in matrix terms. Based on these two task granularities, we present two parallel algorithms for the Mttkrp operation. We address the computational load balance and communication cost reduction problems for the two algorithms, and present hypergraph partitioning-based models

to tackle these problems with the off-the-shelf partitioning tools.

Once the MTTKRP operation is efficiently parallelized, the other parts of the CP-ALS algorithm are straightforward. Nonetheless, we design a library for the parallel CP-ALS algorithm to test the proposed MTTKRP algorithms and report scalability results up to 1024 MPI ranks.

3.1 Background and notation

3.1.1 Hypergraph partitioning

Let $H = (V, E)$ be a hypergraph with the vertex set V and hyperedge set E. Let us associate weights with the vertices and costs with the hyperedges: $w(v)$ denotes the weight of a vertex v and c_h denotes the cost of a hyperedge. For a given integer $K \geq 2$, a K-way vertex partition of a hypergraph $H = (V, E)$ is denoted as $\Pi = \{V_1, \ldots, V_K\}$, where (i) the parts are non-empty; (ii) mutually exclusive, $V_k \cap V_\ell = \emptyset$ for $k \neq \ell$; and (iii) collectively exhaustive, $V = \bigcup V_k$.

Let $w(V_k) = \sum_{v \in V_k} w_v$ be the total weight of the vertices in the set V_k and $w_{avg} = w(V)/K$ be the average part weight. If each part $V_k \in \Pi$ satisfies the *balance criterion*

$$w(V_k) \leq w_{avg}(1 + \varepsilon), \quad \text{for } k = 1, 2, \ldots, K \tag{3.1}$$

we say that Π is *balanced* where ε represents the maximum allowed imbalance ratio.

In a partition Π, a hyperedge that has at least one vertex in a part is said to *connect* that part. The number of parts connected by a hyperedge h, i.e., *connectivity*, is denoted as λ_h. Given a vertex partition Π of a hypergraph $H = (V, E)$, one can measure the size of the cut induced by Π as

$$\chi(\Pi) = \sum_{h \in E} \mathsf{c}_h(\lambda_h - 1) \,. \tag{3.2}$$

This cut measure is called the *connectivity-1* cutsize metric.

Given $\varepsilon > 0$ and an integer $K > 1$, the standard hypergraph partitioning problem is defined as the task of finding a balanced partition Π with K parts such that $\chi(\Pi)$ is minimized. The hypergraph partitioning problem is NP-hard [150].

A variant of the above problem is the *multi-constraint hypergraph partitioning* [131]. In this variant, each vertex has an associated vector of weights. The partitioning objective is the same as above, and the partitioning constraint is to satisfy a balancing constraint for each weight. Let $w(v, i)$ denote the C weights of a vertex v for $i = 1, \ldots, C$. In this variant, the balance criterion (3.1) is rewritten as

$$w(V_k, i) \leq w_{avg,i}\,(1 + \varepsilon) \text{ for } k = 1, \ldots, K \text{ and } i = 1, \ldots, C \tag{3.3}$$

where the ith weight $w(V_k, i)$ of a part V_k is defined as the sum of the ith weights of the vertices in that part, $w_{avg,i}$ is the average part weight for the ith weight of all vertices, and ε represents the allowed imbalance ratio.

3.1.2 Tensor operations

In this chapter, we use N to denote the number of dimensions (the order) of a tensor. For the sake of simplicity, we describe all the notation and the algorithms for $N = 3$, even though our algorithms and implementations have no such restriction. We explicitly generalize the discussion to general order-N tensors whenever we find necessary. A *fiber* in a tensor is defined by fixing every index but one, e.g., if $\mathcal{X}$ is a third-order tensor, $\mathcal{X}_{:,j,k}$ is a mode-1 fiber and $\mathcal{X}_{i,j,:}$ is a mode-3 fiber. A *slice* in a tensor is defined by fixing only one index, e.g., $\mathcal{X}_{i,:,:}$, refers to the ith slice of $\mathcal{X}$ in mode 1. We use $|\mathcal{X}_{i,:,:}|$ to denote the number of nonzeros in $\mathcal{X}_{i,:,:}$.

Tensors can be *matricized* in any mode. This is achieved by identifying a subset of the modes of a given tensor $\mathcal{X}$ as the rows and the other modes of $\mathcal{X}$ as the columns of a matrix and appropriately mapping the elements of $\mathcal{X}$ to those of the resulting matrix. We will be exclusively dealing with the matricizations of tensors along a single mode. For example, take $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. Then $\mathbf{X}_{(1)}$ denotes the mode-1 matricization of $\mathcal{X}$ in such a way that the rows of $\mathbf{X}_{(1)}$ corresponds to the first mode of $\mathcal{X}$ and the columns corresponds to the remaining modes. The tensor element $x_{i_1,\ldots,i_N}$ corresponds to the element $\left(i_1, 1 + \sum_{j=2}^{N} \left[(i_j - 1) \prod_{k=1}^{j-1} I_k\right]\right)$ of $\mathbf{X}_{(1)}$. Specifically, each column of the matrix $\mathbf{X}_{(1)}$ becomes a mode-1 fiber of the tensor $\mathcal{X}$. Matricizations in the other modes are defined similarly.

For two matrices $\mathbf{A}_{I_1 \times J_1}$ and $\mathbf{B}_{I_2 \times J_2}$, the *Kronecker product* is

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{1,1}\mathbf{B} & \cdots & a_{1,J_1}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{I_1,1}\mathbf{B} & \cdots & a_{I_1,J_1}\mathbf{B} \end{bmatrix} .$$

For two matrices $\mathbf{A}_{I_1 \times J}$ and $\mathbf{B}_{I_2 \times J}$, the *Khatri-Rao product* is

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} \mathbf{A}(:,1) \otimes \mathbf{B}(:,1) & \cdots & \mathbf{A}(:,J) \otimes \mathbf{B}(:,J) \end{bmatrix} ,$$

which is of size $I_1 I_2 \times J$.

For two matrices $\mathbf{A}_{I \times J}$ and $\mathbf{B}_{I \times J}$, the *Hadamard product* is

$$\mathbf{A} * \mathbf{B} = \begin{bmatrix} a_{1,1}b_{1,1} & \cdots & a_{1,J}b_{1,J} \\ \vdots & \ddots & \vdots \\ a_{I,1}b_{I,1} & \cdots & a_{I,J}b_{I,J} \end{bmatrix} .$$

Recall that the CP-decomposition of rank R of a given tensor $\mathcal{X}$ factorizes $\mathcal{X}$ into a sum of R rank-one tensors. For $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, it yields $\mathcal{X} \approx$

$\sum_{r=1}^{R} a_r \circ b_r \circ c_r$, for $a_r \in \mathbb{R}^I$, $b_r \in \mathbb{R}^J$ and $c_r \in \mathbb{R}^K$, where $\circ$ is the outer product of the vectors. Here the matrices $\mathbf{A} = [a_1, \ldots, a_R]$, $\mathbf{B} = [b_1, \ldots, b_R]$, and $\mathbf{C} = [c_1, \ldots, c_R]$ are called the factor matrices, or factors. For N-mode tensors, we use $\mathbf{U}_1, \ldots, \mathbf{U}_N$ to refer to the factor matrices.

We revise the Alternating Least Squares (ALS) method for obtaining a rank-R approximation of a tensor $\mathcal{X}$ with the CP-decomposition [38, 107]. A common formulation of CP-ALS is shown in Algorithm 1 for third order tensors. At each iteration, each factor matrix is recomputed while fixing the other two; e.g., $\mathbf{A} \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$. This operation is performed in the following order: $\mathbf{M}_A = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$, $\mathbf{V} = (\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$, and then $\mathbf{A} \leftarrow \mathbf{M}_A\mathbf{V}$. Here $\mathbf{V}$ is a dense matrix of size $R \times R$ and is easy to compute. The important issue is the efficient computation of the MTTKRP operations yielding $\mathbf{M}_A$, similarly $\mathbf{M}_B = \mathbf{X}_{(2)}(\mathbf{A} \odot \mathbf{C})$ and $\mathbf{M}_C = \mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})$.

Algorithm 1: CP-ALS for the 3rd order tensors

Input : $\mathcal{X}$: A 3rd order tensor
R: The rank of approximation
Output: CP decomposition $[\![\boldsymbol{\lambda}; \mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$

1 **repeat**
2 $\quad \mathbf{A} \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$
3 $\quad$ Normalize columns of $\mathbf{A}$
4 $\quad \mathbf{B} \leftarrow \mathbf{X}_{(2)}(\mathbf{C} \odot \mathbf{A})(\mathbf{A}^T\mathbf{A} * \mathbf{C}^T\mathbf{C})^\dagger$
5 $\quad$ Normalize columns of $\mathbf{B}$
6 $\quad \mathbf{C} \leftarrow \mathbf{X}_{(3)}(\mathbf{B} \odot \mathbf{A})(\mathbf{A}^T\mathbf{A} * \mathbf{B}^T\mathbf{B})^\dagger$
7 $\quad$ Normalize columns of $\mathbf{C}$ and store the norms as $\boldsymbol{\lambda}$
8 **until** *no improvement or maximum iterations reached*

The sheer size of the Khatri-Rao products makes them impossible to compute explicitly; hence, efficient MTTKRP algorithms find other means to carry out the MTTKRP operation.

3.2 Related work

SPLATT [206] is an efficient implementation of the MTTKRP operation for sparse tensors on shared memory systems. SPLATT implements the MTTKRP operation based on the slices of the dimension in which the factor is updated, e.g., on the mode-1 slices when computing $\mathbf{A} \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$. Nonzeros of the fibers in a slice are multiplied with the corresponding rows of $\mathbf{B}$ and the results are accumulated to be later scaled with the corresponding row of $\mathbf{C}$ to compute the row of $\mathbf{A}$ corresponding to the slice. Parallelization is done using OpenMP directives, and the load balance (in terms of the number of nonzeros in the slices of the mode for

which MTTKRP is computed) is achieved by using the dynamic scheduling policy. Hypergraph models are used to optimize cache performance by reducing the number of times a row of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are accessed. Smith et al. [206] also use N-partite graphs to reorder the tensors for all dimensions. A newer version of SPLATT [204, 205] has a medium grain task definition and a distributed memory implementation.

GigaTensor [123] is an implementation of CP-ALS which follows the Map-Reduce paradigm. All important steps (the MTTKRPs and the computations $\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C}$) are performed using this paradigm. A distinct advantage of GigaTensor is that thanks to Map-Reduce, the issues of fault-tolerance, load balance, and out of core tensor data are automatically handled. The presentation [123] of GigaTensor focuses on three-mode tensors and expresses the map and the reduce functions for this case. Additional map and reduce functions are needed for higher order tensors.

DFacTo [47] is a distributed memory implementation of the MTTKRP operation. It performs two successive sparse matrix-vector multiplies (SpMV) to compute a column of the product $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$. A crucial observation (made also elsewhere [142]) is that this operation can be implemented as $\mathbf{X}_{(2)}^T\mathbf{B}(:, r)$, which can be reshaped into a matrix to be multiplied with $\mathbf{C}(:, r)$ to form the rth column of the MTTKRP. Although SpMV is a well-investigated operation, there is a peculiarity here: the result of the first SpMV forms the values of the sparse matrix used in the second one. Therefore, there are sophisticated data dependencies between the two SpMVs. Notice that DFacTo is rich in SpMV operations: there are two SpMVs per factorization rank per dimension of the input tensor. DFacTo needs to store the tensor matricized in all dimensions, i.e., $\mathbf{X}_{(1)}, \ldots, \mathbf{X}_{(N)}$. In low dimensions, this can be a slight memory overhead; yet in higher dimensions the overhead could be non-negligible. DFacTo uses MPI for parallelization yet fully stores the factor matrices in all MPI ranks. The rows of $\mathbf{X}_{(1)}, \ldots, \mathbf{X}_{(N)}$ are blockwise distributed (statically). With this partition, each process computes the corresponding rows of the factor matrices. Finally, DFacTo performs an MPI_Allgatherv operation to communicate the new results to all processes, which results in $(I_n/P)\log_2 P$ communication volume per process (assuming a hypercube algorithm) when computing the n^{th} factor matrix having I_n rows using P processes.

Tensor Toolbox [18] is a MATLAB toolbox for handling tensors. It provides many essential operations and enables fast and efficient realizations of complex algorithms in MATLAB for sparse tensors [17]. Among those operations, MTTKRP implementations are provided and used in CP-ALS method. Here, each column of the output is computed by performing $N-1$ sparse tensor vector multiplications. Another well-known MATLAB toolbox is the N-way toolbox [11] which is essentially for dense tensors and incorporates now support for sparse tensors [2] through Tensor Toolbox. Tensor

Toolbox and the related software provide excellent means for rapid prototyping of algorithms and also efficient programs for tensor operations that can be handled within MATLAB.

3.3 Parallelization

A common approach in implementing the MTTKRP is to explicitly matricize a tensor across all modes, and then perform the Khatri-Rao product using the matricized tensors [47, 206]. Matricizing a tensor in a mode i requires column index values up to $\prod_{k \neq i}^{N} I_k$, which can exceed the integer value limits supported by modern architectures when using tensors of higher order and very large dimensions. Also, matricizing across all modes results in N replications of a tensor, which can exceed the memory limitations. Hence, in order to be able to handle large tensors we store them in coordinate format for the MTTKRP operation, which is also the method of choice in Tensor Toolbox [18]. In this format, each nonzero value is stored along with all of its position indices.

With a tensor stored in the coordinate format, MTTKRP operation can be performed as shown in Algorithm 2. As seen on Line 4 of this algorithm, a row of $\mathbf{B}$ and a row of $\mathbf{C}$ are retrieved, and their Hadamard product is computed and scaled with a tensor entry to update a row of $\mathbf{M}_A$. In general, for an N-mode tensor

$$\mathbf{M}_{U_1}(i_1,:) \leftarrow \mathbf{M}_{U_1}(i_1,:) + x_{i_1,i_2\ldots,i_N}[\mathbf{U}_2(i_2,:) * \cdots * \mathbf{U}_N(i_N,:)]$$

is computed. Here, indices of the corresponding rows of the factor matrices and $\mathbf{M}_{U_1}$ coincide with indices of the unique tensor entry of the operation.

Algorithm 2: MTTKRP for the 3rd order tensors

Input : $\mathcal{X}$: tensor
$\mathbf{B}, \mathbf{C}$: Factor matrices in all modes except the first
I_A: Number of rows of the factor $\mathbf{A}$
R: Rank of the factors
Output: $\mathbf{M}_A = \mathbf{X}_{(1)}(\mathbf{B} \odot \mathbf{C})$

1 Initialize $\mathbf{M}_A$ to zeros of size $I_A \times R$
2 **foreach** $x_{i,j,k} \in \mathcal{X}$ **do**
4 $\quad \mathbf{M}_A(i,:) \leftarrow \mathbf{M}_A(i,:) + x_{i,j,k}[\mathbf{B}(j,:) * \mathbf{C}(k,:)]$

As factor matrices are accessed row-wise, we define computational units in terms of the rows of factor matrices. It follows naturally to partition all factor matrices row-wise and use the same partition for the MTTKRP operation for each mode of an input tensor across all CP-ALS iterations to prevent extra communication. A crucial issue is the task definitions, as this

pertains to the issues of load balancing and communication. We identify a *coarse-grain* and a *fine-grain* task definition.

In the *coarse-grain* task definition, the ith atomic task consists of computing the row $\mathbf{M}_A(i,:)$ using the nonzeros in the tensor slice $\mathcal{X}_{i,:,:}$ and the rows of $\mathbf{B}$ and $\mathbf{C}$ corresponding to the nonzeros in that slice. The input tensor $\mathcal{X}$ does not change throughout the iterations of tensor decomposition algorithms; hence it is viable to make the whole slice $\mathcal{X}_{i,:,:}$ available to the process holding $\mathbf{M}_A(i,:)$ so that the MTTKRP operation can be performed by only communicating the rows of $\mathbf{B}$ and $\mathbf{C}$. Yet, as CP-ALS requires the MTTKRP in all modes, and each nonzero $x_{i,j,k}$ belongs to slices $\mathcal{X}(i,:,:)$, $\mathcal{X}(:,j,:)$, and $\mathcal{X}(:,:,k)$, we need to replicate tensor entries in the owner processes of these slices. This may require up to N times replication of the tensor, depending on its partitioning. Note that an explicit matricization always requires exactly N replications of tensor entries.

In the *fine-grain* task definition, an atomic task corresponds to the multiplication of a tensor entry with the Hadamard product of the corresponding rows of $\mathbf{B}$ and $\mathbf{C}$. Here, tensor nonzeros are *partitioned* among processes with no replication to induce a task partition by following the owner-computes rule. This necessitates communicating the rows of $\mathbf{B}$ and $\mathbf{C}$ that are needed by these atomic tasks. Furthermore, partial results on the rows of $\mathbf{M}_A$ need to be communicated, as without duplicating tensor entries, we cannot in general compute all contributions to a row of $\mathbf{M}_A$. Here, the partition of $\mathcal{X}$ should be useful in all modes, as the CP-ALS method requires the MTTKRP in all modes.

The coarse-grain task definition resembles the one-dimensional (1D) row-wise (or column-wise) partitioning of sparse matrices, whereas the fine-grain one resembles the two-dimensional (nonzero-based) partitioning of sparse matrices for parallel sparse matrix-vector multiply (SpMV) operations. As is confirmed for SpMV in modern applications, 1D partitioning usually leads to harder problems of load balancing and communication cost reduction. The same phenomenon is likely to be observed in tensors as well. Nonetheless, we cover the coarse-grain task definition, as it is used in the state of the art parallel MTTKRP [47, 206] methods, which partition the input tensor by slices.

3.3.1 Coarse-grain task model

In the coarse-grain task model, computing the rows of $\mathbf{M}_A$, $\mathbf{M}_B$, and $\mathbf{M}_C$ are defined as the *atomic tasks*. Let μ_A denote the partition of the first mode's indices among the processes, i.e., $\mu_A(i) = p$, if the process p is responsible for computing $\mathbf{M}_A(i,:)$. Similarly, let μ_B and μ_C define the partition of the second and the third mode indices. The process owning $\mathbf{M}_A(i,:)$ needs the entire tensor slice $\mathcal{X}_{i,:,:}$; similarly, the process owning $\mathbf{M}_B(j)$ needs $\mathcal{X}_{:,j,:}$ and the owner of $\mathbf{M}_C(k,:)$ needs $\mathcal{X}_{:,:,k}$. This necessitates duplication of some

tensor nonzeros to prevent unnecessary communication.

One needs to take the context of CP-ALS into account when parallelizing the MTTKRP operation. First, the output $\mathbf{M}_A$ of MTTKRP is transformed into $\mathbf{A}$. Since $\mathbf{A}(i,:)$ is computed simply by multiplying $\mathbf{M}_A(i,:)$ with the matrix $(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$, we make the process which owns $\mathbf{M}_A(i,:)$ responsible for computing $\mathbf{A}(i,:)$. Second, N MTTKRP operations follow one another in an iteration. Assuming that every process has the required rows of the factor matrices while executing MTTKRP for the first mode, it is advisable to implement the MTTKRP in such a way that its output $\mathbf{M}_A$, after transformed into $\mathbf{A}$, is communicated. This way, all processes will have the necessary data for executing the MTTKRP for the next mode. With these in mind, the coarse-grain parallel MTTKRP method executes Algorithm 3 at each process p.

Algorithm 3: Coarse-grain MTTKRP for the first mode of third order tensors at process p within CP-ALS

Input : I_p, indices where $\mu_A(i) = p$
$\mathcal{X}_{I_p,:,:}$, tensor slices
$\mathbf{B}^T\mathbf{B}$ and $\mathbf{C}^T\mathbf{C}$, $R \times R$ matrices
$\mathbf{B}(J_p,:), \mathbf{C}(K_p,:)$, Rows of the factor matrices, where J_p and K_p correspond to the unique second and third mode indices in $\mathcal{X}_{I_p,:,:}$

1 **On exit** : $\mathbf{A}(I_p,:)$ is computed, its rows are sent to processes needing them; $\mathbf{A}^T\mathbf{A}$ is available

2 Initialize $\mathbf{M}_A(I_p,:)$ to all zeros of size $|I_p| \times R$.
3 **foreach** $i \in I_p$ **do**
4 **foreach** $x_{i,j,k} \in \mathcal{X}_{i,:,:}$ **do**
6 $\mathbf{M}_A(i,:) \leftarrow \mathbf{M}_A(i,:) + x_{i,j,k}[\mathbf{B}(j,:) * \mathbf{C}(k,:)]$
8 $\mathbf{A}(I_p,:) \leftarrow \mathbf{M}_A(I_p,:)(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$
9 **foreach** $i \in I_p$ **do**
11 Send $\mathbf{A}(i,:)$ to all processes having nonzeros in $\mathcal{X}_{i,:,:}$.
12 Receive $\mathbf{A}(i,:)$ from $\mu_A(i)$ for each owned nonzero $x_{i,j,k}$.
14 Locally compute $\mathbf{A}(I_p,:)^T\mathbf{A}(I_p,:)$ and all-reduce the results to form $\mathbf{A}^T\mathbf{A}$.

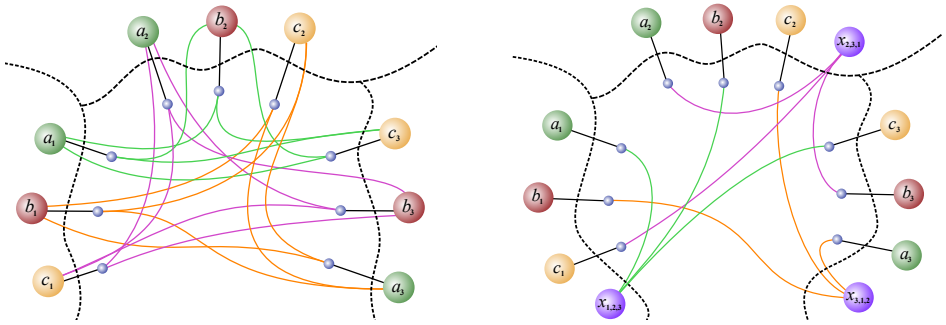
As seen in Algorithm 3, the process p computes $\mathbf{M}_A(i,:)$ for all i with $\mu_A(i) = p$ on Line 6. This is possible without any communication, if we assume the preconditions that $\mathbf{B}^T\mathbf{B}$, $\mathbf{C}^T\mathbf{C}$, and the required rows of $\mathbf{B}$ and $\mathbf{C}$ are available. We need to satisfy this precondition for the MTTKRP operations in the remaining modes. Once $\mathbf{M}_A(I_p,:)$ is computed, it is multiplied on Line 8 with $(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$ to obtain $\mathbf{A}(I_p,:)$. Then, the process p sends the rows of $\mathbf{A}(I_p,:)$ to other processes who will need them, then receive the ones that it will need. More precisely, the process p sends $\mathbf{A}(i,:)$ to the process r, where $\mu_B(j) = r$ or $\mu_C(k) = r$ for a nonzero $x_{i,j,k} \in \mathcal{X}$, thereby satisfying the stated precondition for the remaining MTTKRP op-

erations. Since computing $\mathbf{A}(i,:)$ required $\mathbf{B}^T\mathbf{B}$ and $\mathbf{C}^T\mathbf{C}$ to be available at each process, we also need to compute $\mathbf{A}^T\mathbf{A}$ and make the result available to all processes. We perform this by computing local multiplications $\mathbf{A}(I_p,:)^T\mathbf{A}(I_p,:)$ and all-reducing the partial results (Line 14).

We now investigate the problems of obtaining load balance and reducing the communication cost in an iteration of CP-ALS given in Algorithm 3; that is, when Algorithm 3 is applied N times, once for each mode. Line 6 is the computational core; the process p performs $\sum_{i\in I_p} |\mathcal{X}_{i,:,:}|$ many Hadamard products (of vectors of size R) without any communication. In order to achieve load balance, one should partition the slices of the first mode equitably by taking the number of nonzeros into account. Line 8 does not involve communication, where load balance can be achieved by assigning almost equal number of slices to the processes. Line 14 requires an almost equal number of slices to the processes for load balance. The operations at Lines 8 and 14 can be performed very efficiently using BLAS3 routines; therefore, their costs are generally negligible in compare to the cost of the sparse irregular computations at Line 6. There are two lines, 11 and 14, requiring communication. Line 14 requires the collective all-reduce communication, and one can rely on efficient MPI implementations. The communication at Line 11 is irregular and would be costly. Therefore, the problems of computational load balance and communication cost need to be addressed with regards to Lines 6 and 11, respectively.

Let a_i be the task of computing $\mathbf{M}_A(i,:)$ at Line 6. Let also T_A be the set of tasks a_i for $i = 1,\ldots,I_A$, and b_j, c_k, T_B, and T_C be defined similarly. In the CP-ALS algorithm, there are dependencies among the triplets a_i, b_j, and c_k of the tasks for each nonzero $x_{i,j,k}$. Specifically, the task a_i depends on the tasks b_j and c_k. Similarly, b_j depends on the tasks a_i and c_k, and c_k depends on the tasks a_i and b_j. These dependencies are the source of the communication at Line 11; e.g., $\mathbf{A}(i,:)$ should be sent to the processes that own $\mathbf{B}(j,:)$ and $\mathbf{C}(k,:)$ for the each nonzero $x_{i,j,k}$ in the slice $\mathcal{X}_{i,:,:}$. These dependencies can be modeled using graphs or hypergraphs by identifying the tasks with vertices, and their dependencies with edges and hyperedges, respectively. As is well-known in similar parallelization contexts [39, 111, 112], the standard graph partition related metric of "edge cut" would loosely relate to the communication cost. We therefore propose a hypergraph model for modeling the communication and computational requirements of the MTTKRP operation in the context of CP-ALS.

For a given tensor $\mathcal{X}$, we build the d-partite, d-uniform hypergraph $H = (V,E)$ described in Section 1.5. In this model, the vertex set V corresponds to the set of tasks. We abuse the notation and use a_i to refer to a task and its corresponding vertex in V. We then set $V = T_A \cup T_B \cup T_C$. Since one needs load balance on Line 6 for each mode, we use vectors of size N for vertex weights. We set $w(a_i,1) = |\mathcal{X}_{i,:,:}|$, $w(b_j,2) = |\mathcal{X}_{:,j,:}|$, and $w(c_k,3) = |\mathcal{X}_{:,:,k}|$



(a) Coarse-grain model (b) Fine-grain model

Figure 3.1: Coarse and fine-grain hypergraph models for the $3 \times 3 \times 3$ tensor $\mathcal{X} = \{(1,2,3),(2,3,1),(3,1,2)\}$.

as the vertex weights in the indicated modes, and assign weight 0 in all other modes. The dependencies causing communication at Line 11 are one-to-many: for a nonzero $x_{i,j,k}$, any one of a_i, b_j, and c_k is computed by using the data associated with the other two. Following the guidelines in building the hypergraph models [209], we add a hyperedge corresponding to each task (or each row of the factor matrices) in $T_A \cup T_B \cup T_C$, in order to model the data dependencies to that specific task. We use n_i^a to denote the hyperedge corresponding to the task a_i, and define n_j^b and n_k^c similarly. For each nonzero $x_{i,j,k}$, we add the vertices a_i, b_j, and c_k to the hyperedges n_i^a, n_j^b, and n_k^c. Let J_i and K_i be all unique 2nd and 3rd mode indices, respectively, of the nonzeros in $\mathcal{X}_{i,:,:}$. Then, the hyperedge n_i^a contains the vertex a_i, all vertices b_j for $j \in J_i$, and all vertices c_k for $k \in K_i$.

Figure 3.1a demonstrates the hypergraph model for the coarse-grain task decomposition of a sample $3 \times 3 \times 3$ tensor with nonzeros $(1,2,3)$, $(2,3,1)$, and $(3,1,2)$. Labels for the hyperedges, shown as small blue circles, are not provided; yet it should be clear from the context, e.g., $a_i \in n_i^a$ is shown with a non-curved black line between the vertex and the hyperedge. For the first, second, and third nonzeros, we add the corresponding vertices to related hyperedges, which are shown with green, magenta, and orange curves, respectively. It is interesting to note that the N-partite graph of Smith et al. [206] and this hypergraph model are related; their adjacency structure when laid as a matrix give the same matrix.

Consider now a P-way partition of the vertices of $H = (V, E)$ and the association of each part with a unique process for obtaining a P-way parallel CP-ALS. The task a_i incurs $|\mathcal{X}_{i,:,:}|$ Hadamard products in the first mode in the process $\mu_A(i)$, and has no other work when computing the MTTKRPs $\mathbf{M}_B = \mathbf{X}_{(2)}(\mathbf{A} \odot \mathbf{C})$ and $\mathbf{M}_C = \mathbf{X}_{(3)}(\mathbf{A} \odot \mathbf{B})$ in the other modes. This observation (combined with the corresponding one for b_js and c_ks) shows that any

partition satisfying the balance condition for each weight component (3.3) corresponds to a balanced partition of Hadamard products (Line 6) in each mode. Now consider the messages, which are of size R, sent by the process p regarding $\mathbf{A}(i,:)$. These messages are needed by the processes that have slices in modes 2 and 3 that have nonzeros whose first mode index is i. These processes correspond to the parts connected by the hyperedge n_i^a. For example, in Figure 3.1*a* due to the nonzero $(3,1,2)$, a_3 has a dependency to b_1 and c_2, which are modeled by the curves from a_3 to n_1^b and n_2^c. Since a_3, b_1, and c_2 reside in different parts, a_3 contributes one to the connectivity of n_1^b and n_2^c, which accurately captures the total communication volume by increasing the total connectivity-1 metric by two.

Notice that the part $\mu_A(i)$ is always connected by n_i^a—assuming that the slice $\mathcal{X}_{i,:,:}$ is not empty. Therefore, minimizing the connectivity-1 metric (3.2) *exactly* amounts to minimizing the total communication volume required for MTTKRP in all computational phases of a CP-ALS iteration, if we set the cost $\mathsf{c}_h = R$ for each hyperedge h.

3.3.2 Fine-grain task model

In the fine-grain task model we assume a *partition* of the nonzeros of the tensor; hence there is no duplication. Let $\mu_{\mathcal{X}}$ denote the partition of the tensor nonzeros, e.g., $\mu_{\mathcal{X}}(i,j,k) = p$ if the process p holds $x_{i,j,k}$. Whenever $\mu_{\mathcal{X}}(i,j,k) = p$, the process p performs the Hadamard product of $\mathbf{B}(j,:)$ and $\mathbf{C}(k,:)$ in the first mode, $\mathbf{A}(i,:)$ and $\mathbf{C}(k,:)$ in the second mode, and $\mathbf{A}(i,:)$ and $\mathbf{B}(j,:)$ in the third mode; then scales the result with $x_{i,j,k}$. Let again μ_A, μ_B, and μ_C denote the partition on the indices of the corresponding modes; that is, $\mu_A(i) = p$ denotes that the process p is responsible for computing $\mathbf{M}_A(i,:)$. As in the coarse-grain model, we take the CP-ALS context into account while designing the MTTKRP algorithm: (i) the process which is responsible for $\mathbf{M}_A(i,:)$ is also held responsible for $\mathbf{A}(i,:)$; (ii) at the beginning, all rows of the factor matrices $\mathbf{B}(j,:)$ and $\mathbf{C}(k,:)$ where $\mu_{\mathcal{X}}(i,j,k) = p$ are available to the process p; (iii) at the end, all processes get the $R \times R$ matrix $\mathbf{A}^T\mathbf{A}$; and (iv) each process has all the rows of $\mathbf{A}$ that are required for the upcoming MTTKRP operations in the second or the third modes. With these in mind, the fine-grain parallel MTTKRP method executes Algorithm 4 at each process p.

In Algorithm 4, $\mathcal{X}_p$ and I_p denote the set of nonzeros and the set of first mode indices assigned to the process p. As seen in the algorithm, the process p has the required data to carry out all Hadamard products corresponding to all $x_{i,j,k} \in \mathcal{X}_p$ on Line 5. After scaling by $x_{i,j,k}$, the Hadamard products are locally accumulated in $\mathbf{M}_A$, which contains a row for all i where $\mathcal{X}_p$ has a nonzero on the slice $\mathcal{X}_{i,:,:}$. Notice that a process generates a partial result for each slice of the first mode in which it has a nonzero. The relevant row indices (the set of unique first mode indices in $\mathcal{X}_p$) are denoted by F_p.

Algorithm 4: Fine-grain MTTKRP for the first mode of 3rd order tensors at process p within CP-ALS

Input : $\mathcal{X}_p$, tensor nonzeros where $\mu_{\mathcal{X}}(i,j,k) = p$
I_p, the first mode indices where $\mu_A(I_p) = p$
F_p, the set of unique first mode indices in $\mathcal{X}_p$
$\mathbf{B}^T\mathbf{B}$ and $\mathbf{C}^T\mathbf{C}$: $R \times R$ matrices
$\mathbf{B}(J_p,:), \mathbf{C}(K_p,:)$: Rows of the factor matrices, where J_p and K_p correspond to the unique second and third mode indices in $\mathcal{X}_p$

On exit : $\mathbf{A}(I_p,:)$ is computed, its rows are sent to processes needing them; $\mathbf{A}(F_p,:)$ is updated; and $\mathbf{A}^T\mathbf{A}$ is available

Initialize $\mathbf{M}_A(F_p,:)$ to all zeros of size $|F_p| \times R$
foreach $x_{i,j,k} \in \mathcal{X}_p$ **do**
  $\mathbf{M}_A(i,:) \leftarrow \mathbf{M}_A(i,:) + x_{i,j,k}[\mathbf{B}(j,:) * \mathbf{C}(k,:)]$
foreach $i \in F_p \setminus I_p$ **do**
  Send $\mathbf{M}_A(i,:)$ to $\mu_A(i)$
Receive contributions to $\mathbf{M}_A(i,:)$ for $i \in I_p$ and add them up
$\mathbf{A}(I_p,:) \leftarrow \mathbf{M}_A(I_p,:)(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$
foreach $i \in I_p$ **do**
  Send $\mathbf{A}(i,:)$ to all processes having nonzeros in $\mathcal{X}_{i,:,:}$.
foreach $i \in F_p \setminus I_p$ **do**
  Receive $\mathbf{A}(i,:)$ from $\mu_A(i)$
Locally compute $\mathbf{A}(I_p,:)^T\mathbf{A}(I_p,:)$ and all-reduce the results to form $\mathbf{A}^T\mathbf{A}$

Some indices in F_p are not owned by the process p. For such an $i \in F_p \setminus I_p$, the process p sends its contribution to $\mathbf{M}_A(i,;)$ to the owner process $\mu_A(i)$ at Line 8. Again, messages of the process p destined to the same process are combined. Then, the process p receives contributions to $\mathbf{M}_A(i,:)$ where $\mu_A(i) = p$ at Line 10. Each such contribution is accumulated, and the new $\mathbf{A}(i,:)$ is computed by a multiplication with $(\mathbf{B}^T\mathbf{B} * \mathbf{C}^T\mathbf{C})^\dagger$ at Line 12. Finally, the updated $\mathbf{A}$ is communicated to satisfy the precondition of the upcoming MTTKRP operations. Note that the communications at Lines 15 and 18 are the duals of those at Lines 10 and 8, respectively, in the sense that the directions of the messages are reversed and the messages always pertain to the same row indices. For example, $\mathbf{M}_A(i,:)$ is sent to the process $\mu_A(i)$ at Line 8 and $\mathbf{A}(i,:)$ is received from $\mu_A(i)$ at Line 18. Hence, the process p generates partial results for each slice $\mathcal{X}_{i,:,:}$ on which it has a nonzero; either keeps it or sends it to the owner of $\mu_A(i)$ (at Line 8), and if so, receives the updated row $\mathbf{A}(i,:)$ later (at Line 18). Finally, the algorithm finishes by an all-reduce operation to make $\mathbf{A}^T\mathbf{A}$ available to all processes for the upcoming MTTKRP operations.

We now investigate the computational and communication requirements of Algorithm 4. As before, the computational core is at Line 5, where the Hadamard products of the row vectors $\mathbf{B}(j,:)$ and $\mathbf{C}(k,:)$ are computed for each nonzero $x_{i,j,k}$ that a process owns, and the result is added to $\mathbf{M}_A(i,:)$. Therefore, each nonzero $x_{i,j,k}$ incurs three Hadamard products (one per mode) in its owner process $\mu_{\mathcal{X}}(i,j,k)$ in an iteration of the CP-ALS algorithm. Other computations are either efficiently handled with BLAS3 routines (Lines 12 and 20), or they (Line 10) are not as many as the number of Hadamard products. The significant communication operations are the sends at Lines 8 and 15, and the corresponding receives at Lines 10 and 18. Again, the all-reduce operation at Line 20 would be handled efficiently by the MPI implementation. Therefore, the problems of computational load balance and reduced communication cost need to be addressed by partitioning the tensor nonzeros equitably among processes (for load balance at Line 5) while trying to confine all slices of all modes to a small set of processes (to reduce communication at Lines 8 and 18).

We propose a hypergraph model for the computational load and communication volume of the fine-grain parallel MTTKRP operation in the context of CP-ALS. For a given tensor, the hypergraph $H = (V, E)$ with the vertex set V and hyperedge set E is defined as follows. The vertex set V has four types of vertices: a_i, for $i = 1, \ldots, I_A$; b_j, for $j = 1, \ldots, I_B$; c_k for $k = 1, \ldots, I_C$; and the vertex $x_{i,j,k}$ we define for each nonzero in $\mathcal{X}$ by abusing the notation. The vertex $x_{i,j,k}$ represents the Hadamard products $\mathbf{B}(j,:) * \mathbf{C}(k,:)$, $\mathbf{A}(i,:) * \mathbf{C}(k,:)$, and $\mathbf{A}(i,:) * \mathbf{B}(j,:)$ to be performed during the MTTKRP operations at different modes. There are three types of hyperedges in E, and they represent the dependencies of the Hadamard products to the rows of the factor matrices. The hyperedges are defined as follows:

$E = E_A \cup E_B \cup E_C$ where E_A contains a hyperedge n_i^a for each $\mathbf{A}(i,:)$; E_B contains a hyperedge n_j^b for each $\mathbf{B}(j,:)$; and E_C contains a hyperedge n_k^c for each $\mathbf{C}(k,:)$. Initially, n_i^a, n_j^b and n_k^c contain only the corresponding vertices a_i, b_j, and c_k. Then, for each nonzero $x_{i,j,k} \in \mathcal{X}$, we add the vertex $x_{i,j,k}$ to the hyperedges n_i^a, n_j^b and n_k^c to model the data dependency to the corresponding rows of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. As in the previous subsection, one needs a multi-constraint formulation for three reasons. First, since $x_{i,j,k}$ vertices represent the Hadamard products, they should be evenly distributed among the processes. Second, as the owners of the rows of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are possible destinations of partial results (at Line 8), it is advisable to spread them evenly. Third, almost equal number of rows per process implies balanced memory usage and BLAS3 operations at Lines 12 and 20; even though this operation is not our main concern for the computational load. With these constraints, we associate a weight vector of size $N+1$ with each vertex. For a vertex $x_{i,j,k}$ we set $w(x_{i,j,k},:) = [0,0,0,3]$, for vertices a_i, b_j and c_k, we set $w(a_i,:) = [1,0,0,0]$, $w(b_j,:) = [0,1,0,0]$, $w(c_k,:) = [0,0,1,0]$.

In Figure 3.1b, we demonstrate the fine-grain hypergraph model for the sample tensor of the previous section. We exclude the vertex weights from the figure. Similar to the coarse-grain model, dependency on the rows of $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are modeled by connecting each vertex $x_{i,j,k}$ to the hyperedges n_i^a, n_j^b, and n_k^c to capture the communication cost.

Consider now a P-way partition of the vertices of $H = (V,E)$ and associate each part with a unique process to obtain P-way parallel CP-ALS with the fine-grain MTTKRP operation. The partition of the $x_{i,j,k}$ vertices uniquely partitions the Hadamard products. Satisfying the fourth balance constraint in (3.3), balances the computational loads of the processes. Similarly, satisfying the first three constraints leads to an equitable partition of the rows of factor matrices among processes. Let $\mu_{\mathcal{X}}(i,j,k) = p$ and $\mu_A(i) = \ell$; that is, the vertex $x_{i,j,k}$ and a_i are in different parts. Then, the process p sends a partial result on $\mathbf{M}_A(i,:)$ to the process ℓ at Line 8 of Algorithm 4. By looking at these messages carefully, we see that all processes whose corresponding parts are in the connectivity set of n_A^i will send a message to $\mu_A(i)$. Since $\mu_A(i)$ is also in this set, we see that the total volume of send operations concerning $\mathbf{M}_A(i,:)$ at Line 8 is equal to $\lambda_{n_i^a} - 1$. Therefore, the connectivity-1 cut size metric (3.2) over the hyperedges in E_A encodes the total volume of messages sent at Line 8, if we set $\mathsf{c}_h = R$ for all hyperedges un E_A. Since the send operations at Line 15 are duals of the send operations at Line 8, total volume of messages sent at Line 15 for the first mode is also equal to this number. By extending this reasoning to all hyperedges, we see that the cumulative (over all modes) volume of communication is equal to the connectivity-1 cut-size metric (3.2).

The presence of multiple constraints usually causes difficulties for the current state of the art partitioning tools (Aykanat et al.[15] discuss the is-

sue for PaToH [40]). In preliminary experiments, we observed that this was the case for us as well. In an attempt to circumvent this issue, we designed the following alternative. We start with the same hypergraph, and remove the vertices corresponding to the rows of factor matrices. Then use a single constraint partitioning on the resulting hypergraph to partition the vertices of nonzeros. We are then faced with the problem of assigning the rows of factor matrices, given the partition on the tensor nonzeros. Any objective function (relevant to our parallelization context) in trying to solve this problem would likely be a variant of the NP-complete Bin-Packing Problem [93, p.124], as in the SpMV case [27, 208]. Therefore, we heuristically handle this problem, by making the following observations: (i) the modes can be partitioned one at a time; (ii) each row corresponds to a removed vertex for which there is a corresponding hyperedge; (iii) the partition of $x_{i,j,k}$ vertices defines a connectivity set for each hyperedge. This is essentially multi-constraint partitioning which takes advantage of the listed observations. We use connectivity of corresponding hyperedges as *key value* of rows to impose an order. Then, rows are visited in decreasing order of key values. Each visited row is assigned to the least loaded part in the connectivity set of its corresponding hyperedge, if that part is not overloaded. Otherwise, the visited vertex is assigned to the least loaded part at that moment. Note that in the second case we add one to the total communication volume; otherwise the total communication volume obtained by partitioning $x_{i,j,k}$ vertices remains the same. Note also that key values correspond to the volume of receives at Line 10 of Algorithm 4; hence, this method also aims to balance the volume of receives at Line 10, and the volume of dual send operations at Line 15.

3.4 Experiments

We conducted our experiments on the IDRIS Ada cluster, which consists of 304 nodes each having 128 GBs of memory and four 8-core Intel Sandy Bridge E5-4650 processors running at 2.7 GHz. We ran our experiments up to 1024 cores, and assigned one MPI process per core. All codes we used in our benchmarks were compiled using the Intel C++ compiler (version 14.0.1.106) with -O3 option for compiler optimizations, -axAVX,SSE4.2 to enable SSE optimizations whenever possible, and -mkl option to use the Intel MKL library (version 11.1.1) for LAPACK, BLAS and VML (Vector Math Library) routines. We also compiled DFacTo code using the Eigen (version 3.2.4) library [102] with EIGEN_USE_MKL_ALL flag set to ensure that it utilizes the routines provided in the Intel MKL library for the best performance. We used PaToH [40] with default options to partition the hypergraphs. We created all partitions offline, and ran our experiments on these partitioned tensors on the cluster. We do not report timings for partitioning hypergraphs, which is quite costly, for two reasons. First, in most

Tensor	I_1	I_2	I_3	I_4	#nonzeros
netflix	480K	17K	2K	-	100M
NELL-B	3.2M	301	638K	-	78M
deli	530K	17M	2.4M	1K	140M
flickr	319K	28M	1.6M	713	112M

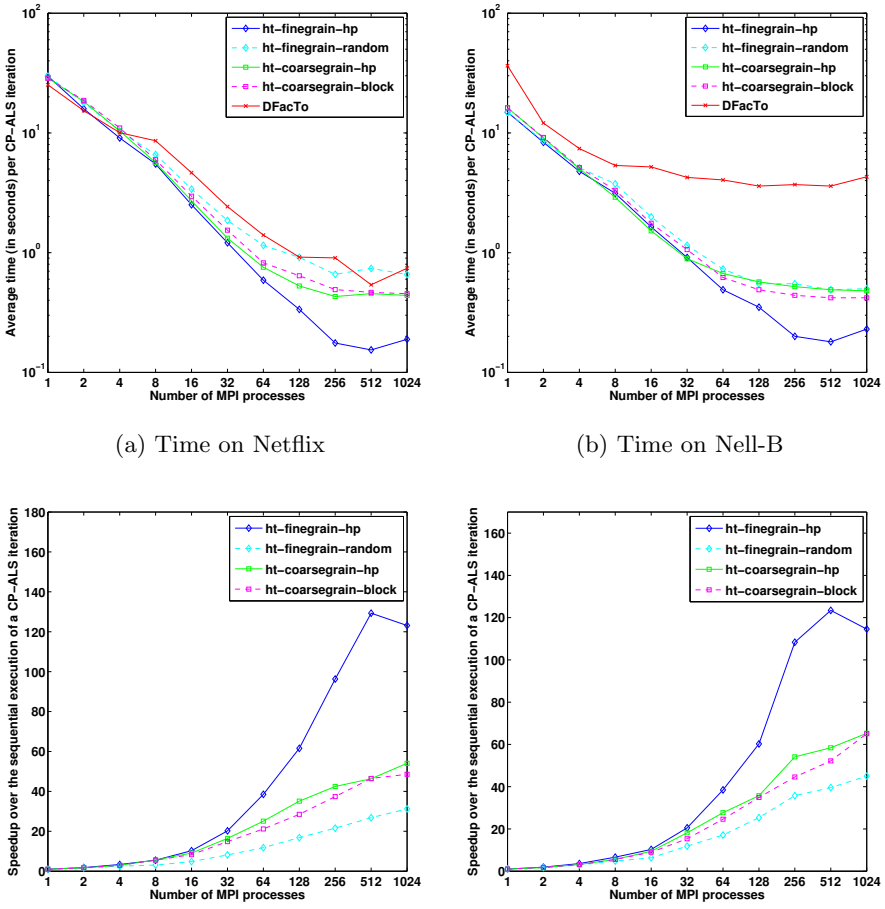
Table 3.1: Size of tensors used in the experiments

applications, the tensors from the real-world data are built incrementally; hence, a partition for the updated tensor can be formed by refining the partition of the previous tensor. Also, one can decompose a tensor multiple times with different ranks of approximation. Thereby, the cost of an initial partitioning of a tensor can be amortized across multiple runs. Second, we had to partition the data on a system different from the one used for CP-ALS runs. It would not be very useful to compare the run time from different systems and repeating the same runs on a different system would not add much to the presented results. We also note that the proposed coarse and fine-grain MTTKRP formulations are independent from the partitioning method.

Our dataset for experiments consists of four sparse tensors, whose sizes are given in Table 3.1 and obtained from various resources [21, 37, 97].

We briefly describe the methods compared in the experiments. **DFacTo** is the CP-ALS routine we compare our methods against, which uses block partitioning of the rows of the matricized tensors to distribute matrix nonzeros equally among processes. In the method **ht-coarsegrain-block**, we operate the coarse-grain CP-ALS implementation in HyperTensor on a tensor whose slices in each mode are partitioned consecutively to ensure equal number of nonzeros among processes, similarly to the DFacTo's approach. The method **ht-coarsegrain-hp** runs the same CP-ALS implementation, and it uses the hypergraph partitioning. The method **ht-finegrain-random** partitions the tensor nonzeros as well as the rows of the factor matrices randomly to establish load balance. It uses the fine-grain CP-ALS implementation in HyperTensor. Finally, the method **ht-finegrain-hp** corresponds to the fine-grain CP-ALS implementation in HyperTensor operating on the tensor partitioned according to the hypergraph model proposed for the fine-grain task decomposition. We benchmark our methods against DFacTo on Netflix and NELL-B datasets only; as DFacTo can only process 3-mode tensors. We let the CP-ALS implementations run for 20 iterations on each data with $R = 10$, and record the average time spent per iteration.

In Figures 3.2a and 3.2b, we show the time spent per CP-ALS iteration on Netflix and NELL-B tensors for all algorithms. In Figures 3.2c and 3.2d, we show the speedups per CP-ALS iteration on Flickr and Delicious tensors for methods excluding DFacTo. While performing the comparison with DFacTo, we preferred to show the time spent per iteration instead of the speedup, as the sequential run time of our methods differs from that of



(a) Time on Netflix

(b) Time on Nell-B

(c) Speedup on Flickr

(d) Speedup on Delicious

Figure 3.2: Time for parallel CP-ALS iteration on Netflix and NELL-B, and speedups on Flickr and Delicious.

DFacTo. Also in Table 3.2, we show the statistics regarding the communication and computation requirements of the 512-way partitions of the Netflix tensor for all proposed methods.

We first observe in Figure 3.2a that on Netflix, ht-finegrain-hp clearly outperforms all other methods by achieving a speedup of 194x with 512 cores over a sequential execution, whereas ht-coarsegrain-hp, ht-coarsegrain-block, DFacTo, and ht-finegrain-random could only yield 69x, 63x, 49x, and 40x speedups, respectively. Table 3.2 shows that on 512-way partitioning of the same tensor, ht-finegrain-random, ht-coarsegrain-block, and ht-coarsegrain-hp result in total send communication volumes of 142M, 80M, and 77M units respectively; whereas ht-finegrain-hp partitioning requires only 7.6M units; which explains the superior parallel performance of ht-finegrain-hp. Similarly in Figures 3.2b, 3.2c, and 3.2d ht-finegrain-hp achieves 81x, 129x, and 123x speedups on NELL-B, Flickr, and Delicious tensors, while the best of all other methods could only obtain 39x, 55x, and 65x, respectively, up to 1024 cores. As seen from these figures, ht-finegrain-hp is the fastest of all proposed methods. It experiences slow-downs at 1024 cores for all instances.

One point to note in Figure 3.2b is that our MTTKRP implementation gets notably more efficient than DFacTo on NELL-B. We believe that the main reason for this difference is due to DFacTo's column-by-column way of computing the factor matrices. We instead compute all columns of the factor matrices simultaneously. This vector-wise mode of operation on the rows of the factors can significantly improve the data locality and cache efficiency which leads to this performance difference. Another point in the same figure is that HyperTensor running with ht-coarsegrain-block partitioning scales better than DFacTo, even though they use similar partitioning of the matricized tensor or the tensor. This is mainly due to the fact that DFacTo uses MPI_Allgatherv to communicate local rows of factor matrices to all processes, which incurs significant increase in the communication volume, whereas our coarse-grain implementation performs point-to-point communications. Finally, in Figure 3.2a, DFacTo has a slight edge over our methods in sequential execution. This could be due to the fact that it requires $3(|\mathcal{X}| + F)$ multiply/add operations across all modes, where F is the average number of non-empty fibers and is upper-bounded by $|\mathcal{X}|$, whereas our implementation always takes $6|\mathcal{X}|$ operations.

Partitioning metrics in the Table 3.2 show that ht-finegrain-hp is able to successfully reduce the total communication volume, which brings about its improved scalability. However, we do not see the same effect when using hypergraph partitioning for the coarse-grain task model, due to inherent limitations of 1D partitionings. Another point to note is that despite reducing the total communication volume explicitly, imbalance in the communication volume still exists among the processes, as this objective is not directly captured by the hypergraph models. However, the most important limitation on further scalability is the communication latency. Table 3.2 shows that

Mode	Comp. load		Comm. volume		Num. msg.	
	Max	Avg	Max	Avg	Max	Avg
ht-finegrain-hp						
1	196672	196251	21079	6367	734	316
2	196672	196251	18028	5899	1022	1016
3	196672	196251	3545	2492	1022	1018
ht-finegrain-random						
1	197507	196251	272326	252118	1022	1022
2	197507	196251	29282	22715	1022	1022
3	197507	196251	7766	4300	1013	1003
ht-coarsegrain-hp						
1	364181	196251	302001	136741	511	511
2	349123	196251	59523	12228	511	511
3	737570	196251	23524	2000	511	507
ht-coarsegrain-block						
1	198602	196251	239337	142006	448	447
2	367966	196251	33889	12458	511	445
3	737570	196251	24659	2049	511	394

Table 3.2: Statistics for the computation and communication requirements in one CP-ALS iteration for 512-way partitionings of the Netflix tensor.

each process communicates with almost all others especially on the small dimensions of the tensors, and the number of messages is doubled for the fine-grain methods. To alleviate this issue, one can try to limit the latency following previous work [201, 208].

3.5 Summary, further notes and references

The pros and cons of the coarse and fine-grain models resemble to those of the 1D and 2D partitionings in the SpMV context. There are two advantages of the coarse-grain model over the fine-grain one. First, there is only one communication/computation step in the coarse-grain model; whereas fine-grain model requires two computation and communication phases. Second, the coarse-grain hypergraph model is smaller than the fine-grain hypergraph model (one vertex per index of each dimension versus additionally having one vertex per nonzero). However, one major limitation with the coarse-grain parallel decomposition is that restricting all nonzeros in an $(N-1)$ dimensional slice of an N dimensional tensor to reside in the same process poses a significant constraint towards communication minimization; an $(N-1)$-dimensional data typically has very large "surface" which necessitates gathering numerous rows of the factor matrices. As N increases, one might expect this phenomenon to increase the communication volume even further. Also, if the tensor $\mathcal{X}$ is not large enough in one of its modes, this poses a granularity problem which potentially causes load imbalance and limits the

parallelism. None of these issues exists in the fine-grain task decomposition.

In a more recent work [135], we extended the discussed approach for efficient parallelization in shared and distributed memory systems. For the shared-memory computations, we proposed a novel computational scheme, called dimension trees, that significantly reduces the computational cost while offering an effective parallelism. We then performed theoretical analyses corresponding to the computational gains and the memory use. These analyses are also validated with the experiments. We also presented comparisons with the recent version of SPLATT [204, 205], which uses a medium grain task model and associated partitioning [187]. Recent work discusses algorithms for partitioning for medium-grain task decomposition [3].

A dimension tree is a data structure that partitions the mode indices of an N-dimensional tensor in a hierarchical manner for computing tensor decompositions efficiently. It was first used in the hierarchical Tucker format representing the hierarchical Tucker decomposition of a tensor [99], which was introduced as a computationally feasible alternative to the original Tucker decomposition for higher order tensors. Our work [135] presents a novel way of using dimension trees for computing the standard CP decomposition of tensors with a formulation that asymptotically reduces the computational cost by memorizing certain partial computations.

For computing the CP decomposition of dense tensors, Phan et al. [189] propose a scheme that divides the tensor modes into two sets, pre-computes the tensor-times-multiple-vector multiplication (TTMVs) for each mode set, and finally reuses these partial results to obtain the final MTTKRP result in each mode. This provides a factor of two improvement in the number of TTMVs over the standard approach, and our dimension tree-based framework can be considered as the generalization of this approach that provides a factor of $N/\log N$ improvement. Li et al. [153] use the same idea of storing intermediate tensors but use a different formulation based on the tensor times tensor multiplication and the tensor–matrix Hadamard product operations for shared memory systems. The overall approach is similar to that proposed by Phan et al. [189], where the difference lies in the application of the method to sparse tensors and auto-tuning to better control the memory use and the gains in the operation counts.

In most of the sparse tensors available (e.g., see FROSTT [203]), one of the dimensions is usually very small. If the indices in that dimension are vertices in a hypergraph, then we have very high degree vertices; if the indices are hyperedges, then we have hyperedges with very large sizes. Both cases are not favorable for the traditional, multilevel hypergraph partitioning methods, where efficient coarsening heuristics' run time are super-linear in terms of vertex/hyperedge degrees.

Part II

Matching and related problems

Chapter 4

Matchings in graphs

In this chapter, we address the problem of approximating the maximum cardinality of a matching in undirected graphs with fast heuristics. Most of the terms and definitions, including the problem itself, is standard, which we summarized in Section 1.2. The formal problem definition is repeated below for convenience.

Approximating the maximum cardinality of a matching in undirected graphs. Given an undirected graph $G = (V, E)$, design a fast (near linear time) heuristic to obtain a matching $\mathcal{M}$ of large cardinality with an approximation guarantee.

We design randomized heuristics for the stated problem and analyze their approximation guarantees. Both the design and analysis of the heuristics are based on doubly stochastic matrices (see Section 1.4), and scaling of nonnegative matrices (in our case the adjacency matrices) to the doubly stochastic form. We therefore first give some background on scaling matrices to doubly stochastic form.

4.1 Scaling matrices to doubly stochastic form

Any nonnegative matrix $\mathbf{A}$ with total support can be scaled with two positive diagonal matrices $\mathbf{D}_R$ and $\mathbf{D}_C$ such that $\mathbf{D}_R\mathbf{A}\mathbf{D}_C$ is doubly stochastic (that is, the sum of entries in any row and in any column of $\mathbf{D}_R\mathbf{A}\mathbf{D}_C$ is equal to one). If the matrix is fully indecomposable, then the scaling matrices are unique up to a scalar multiplication (if the pair $\mathbf{D}_R$ and $\mathbf{D}_C$ scale the matrix, then so $\alpha\mathbf{D}_R$ and $\frac{1}{\alpha}\mathbf{D}_C$ for $\alpha > 0$). If $\mathbf{A}$ has support but not total support, then $\mathbf{A}$ can be scaled to a doubly stochastic matrix but not with two positive diagonal matrices [202]—this fact is also seen in more recent treatments [139, 141, 198]).

The Sinkhorn-Knopp algorithm [202] is a well-known method for scaling matrices to doubly stochastic form. This algorithm generates a sequence of

matrices (whose limit is doubly stochastic) by normalizing the columns and the rows of the sequence of matrices alternately as shown in Algorithm 5. First, the initial matrix is normalized such that each column has sum one. Then, the resulting matrix is normalized so that each row has sum one and so on so forth. If the initial matrix is symmetric and has total support, Sinkhorn-Knopp algorithm will find a symmetric doubly stochastic matrix at the limit; while the intermediate matrices generated in the sequence are not symmetric. Two other iterative scaling algorithms [140, 141, 198] maintain symmetry all along the way, which are therefore more useful for symmetric matrices in practice. Efficient shared memory and distributed memory parallelizations of the Sinkhorn-Knopp and related algorithms have been discussed elsewhere [9, 41, 76].

Algorithm 5: ScaleSK: Sinkhorn-Knopp scaling

Data: $\mathbf{A}$: an $n \times n$ matrix with total support, ε: the error threshold
Result: $\mathbf{d}_r$, $\mathbf{d}_c$: row/column scaling arrays

```
1  for i = 1 to n do
2      d_r[i] ← 1
3      d_c[i] ← 1
4  for j = 1 to n do
5      csum[j] ← Σ_{i∈A_{*j}} a_ij
6  repeat
7      for j = 1 to n do
8          d_c[j] ← 1/csum[j]
9      for i = 1 to n do
10         rsum ← Σ_{j∈A_{i*}} a_ij × d_c[j]
11         d_r[i] ← 1/rsum
12     for j = 1 to n do
13         csum[j] ← Σ_{i∈A_{*j}} d_r[i] × a_ij
14 until |max{1 − csum[j] : 1 ≤ j ≤ n}| ≤ ε
```

4.2 Approximation algorithms for bipartite graphs

Most of the existing heuristics obtain good results in practice, but their worst-case guarantee is only around 1/2. Among those, the Karp-Sipser heuristic [128] is very well known. It finds maximum cardinality matchings in highly sparse (random) graphs, and matches all but $\tilde{O}(n^{1/5})$ vertices of a random undirected graph [14] (this heuristics will be reviewed later in Section 4.2.1). Karp-Sipser also obtains very good results in practice. Currently, it is the suggested one to be used as a jump-start routine [71, 148] for exact algorithms, especially for augmenting-path based ones [132]. Algorithms that achieve an approximation ratio of $1-1/e$, where e is the base

of the natural logarithm are designed for the online case [129]. Many of these algorithms are sequential in nature in that a sequence of greedy decisions are made in the light of the previously made decisions. Another heuristic is obtained by truncating the Hopcroft and Karp (HK) algorithm. HK, starting from a given matching, augments along a maximal set of shortest disjoint paths, until there are no augmenting paths. If one lets HK run until the shortest augmenting paths are of length k, then a $1 - 2/k$ approximate matching is obtained for $k \geq 3$. The run time of this heuristic is $\mathcal{O}(\tau k)$, for a bipartite graph with τ edges.

We propose two matching heuristics (Section 4.2.2) for bipartite graphs. Both heuristics construct a subgraph of the input graph by randomly choosing some edges. They then obtain a maximum matching in the selected subgraph and return it as an approximate matching for the input graph. The probability density function for choosing a given edge in both heuristics is obtained with a matrix scaling method. The first heuristic is shown to deliver a constant approximation guarantee of $1 - 1/e \approx 0.632$ of the maximum cardinality matching under the condition that the scaling method has scaled the input matrix. The second one builds on top of the first one and improves the approximation ratio to 0.866, under the same condition as in the first one. Both of the heuristics are designed to be amenable to parallelization in modern multicore systems. The first heuristic does not require a conflict resolution scheme. Furthermore, it does not have any synchronization requirements. The second heuristic employs Karp-Sipser to find a matching on the selected subgraph. We show that Karp-Sipser becomes an exact algorithm on the selected subgraphs. Further analysis of the properties of those subgraphs is carried out to design a specialized implementation of Karp-Sipser for efficient parallelization. The approximation guarantees of the two proposed heuristics do not deteriorate with the increased degree of parallelization, thanks to their design, which is usually not the case for parallel matching heuristics [16].

For the analysis, we assume that the two vertex classes contain the same number of vertices, and that the associated adjacency matrix has total support. Under these criteria, the Sinkhorn-Knopp scaling algorithm summarized in Section 4.1 converges in a finite number of steps, and obtains a doubly stochastic matrix. Later on (towards the end of Section 4.2.2 and in the experiments presented in Section 4.2.3), we discuss the bipartite graphs without the mentioned properties.

4.2.1 Related work

Parallel (exact) matching algorithms on modern architectures have been recently investigated. Azad et al. [16] study the implementations of a set of known bipartite graph matching algorithms on shared memory systems. Deveci et al. [66, 65] investigate the implementation of some known match-

ing algorithms or their variants on GPU. There are quite good speedups reported in these implementations, yet there are non-trivial instances where parallelism does not help (for any of the algorithms).

Our focus is on matching heuristics that have linear run time complexity and good quality guarantees on the size of the matching. Recent surveys of matching heuristics are given by Duff et al. [71, Section 4], Langguth et al. [148], and more recently by Pothen et al. [195]. Two heuristics, called Greedy and Karp-Sipser, stand out and are suggested as initialization steps in the best two exact matching algorithms [132]. These two heuristics also attracted theoretical interest.

The Greedy heuristic has two variants in the literature. The first variant randomly visits the edges and matches the two endpoints of an edge if they are both available. The theoretical performance guarantee of this heuristic is 1/2, i.e., the heuristic delivers matchings of size at least half of the maximum cardinality of a matching. This is analyzed theoretically [78] and shown to obtain results that are near the worst-case on certain classes of graphs. The second variant of Greedy repeatedly selects a random vertex and matches it with a random neighbor. The matched vertices, along with the ones which become isolated, are removed from the graph and the process continues until the whole graph is consumed. This variant also has a 1/2 worst-case approximation guarantee (see for example a proof by Pothen and Fan [194]), and it is somewhat better ($0.5 + \epsilon$ for $\epsilon \geq 0.0000025$ [13] which has been recently improved to $\epsilon \geq 1/256$ [191]).

We make use of the Karp-Sipser heuristic to design one of the proposed heuristics. We summarize a commonly used variant of Karp-Sipser here and refer the reader to the original paper [128] for details. The theoretical foundation of Karp-Sipser is that if there is a vertex v with exactly one neighbor (v is called *degree-one*), then there is a maximum cardinality matching in which v is matched with its neighbor. That is, matching v with its neighbor is an *optimal* decision. Using this observation, Karp-Sipser runs as follows. Check whether there is a degree-one vertex; if so then match the vertex with its unique neighbor and delete both vertices (and the edges incident on them) from the graph. Continue this way until the graph has no edges (in which case we are done) or all remaining vertices have degree larger than one. In the latter case, pick a random edge, match the two endpoints of this edge, and delete those vertices and the edges incident on them. Then repeat the whole process on the remaining graph. The phase before the first random choice of edges made is called Phase 1, and the rest is called Phase 2 (where new degree-one vertices may arise). The run time of this heuristic is linear. This heuristic matches all but $\tilde{O}(n^{1/5})$ vertices of a random undirected graph [14]. One disadvantage of Karp-Sipser is that because of the degree dependencies of the vertices to the already matched vertices, an efficient parallelism is hard to achieve (a list of degree-one vertices needs to be maintained). That is probably why some inflicted forms (successful but

without any known quality guarantee) of this heuristic were used in recent studies [16].

Recent studies focusing on approximate matching algorithms on parallel systems include heuristics for graph matching problem [42, 86, 105] and also heuristics used for initializing bipartite matching algorithms [16]. Lotker et al. [164] present a distributed $1 - 1/k$ approximate matching algorithm for bipartite graphs. This nice theoretical algorithm has $O(k^3 \log \Delta + k^2 \log n)$ time steps with a message length of $O(\log \Delta)$ where Δ is the maximum degree of a vertex and n is the number vertices in the graph. Blelloch et al. [29] propose an algorithm to compute maximal matchings (1/2 approximate) with $O(\tau)$ *work* and $O(\log^3 \tau)$ *depth* with high probability on a bipartite graph with τ edges. This is an elaboration of the Greedy heuristic for parallel systems. Although the performance metrics *work* and *depth* are quite impressive, the approximation guarantee stays as in the serial variant. A striking property of this heuristic is that it trades parallelism and reduced work while always finding the same matching (including those found in the sequential version). Birn et al. [26] discuss maximal (1/2 approximate) matchings in $O(\log^2 n)$ time and $O(\tau)$ work in the CREW PRAM model. Practical implementations on distributed memory and GPU systems are discussed—a shared memory implementation is left as future work in the cited paper. Patwary et al. [185] discuss an efficient implementation of Karp-Sipser on distributed memory systems and show that the communication requirement (in terms of the volume) is proportional to that of sparse matrix–vector multiplication.

The overall approach is related to a random bipartite graph model called random k-out model [211]. In these graphs, every vertex chooses k neighbors uniformly at random from the other part of vertices. Walkup [211] shows that a 1-out subgraph of a complete bipartite graph $G = (V_1, V_2, E)$ where $|V_1| = |V_2|$ asymptotically almost surely (a.a.s.) does not contain a perfect matching. He also shows that a 2-out subgraph of a complete bipartite graph a.a.s. contains a perfect matching. Karoński and Pittel [125], later with Overman [124], extend the analysis to two other cases. In the first case, the vertices that were not selected at all choose one more neighbor; in the second case the vertices that were selected only once choose one more neighbor. Initial claim [125] was that in the first case, there was a perfect matching a.a.s., which was shown to be false [124]. The existence of perfect matchings in the second case, in which the graph is still between 1-out and 2-out, has been shown recently [124].

4.2.2 Two matching heuristics

We propose two matching heuristics for the approximate maximum cardinality matching problem on bipartite graphs. The heuristics are efficiently parallelizable and have guaranteed approximation ratios. The first heuristic

does not require synchronization nor conflict resolution assuming that the write operations to the memory are atomic (such settings are common; see a discussion in the original paper [76]). This heuristic has an approximation guarantee of around 0.632. The second heuristic is designed to obtain larger matchings compared to those obtained by the first one. This heuristic employs Karp-Sipser on a judiciously chosen subgraph of the input graph. We show that for this subgraph, Karp-Sipser is an exact algorithm, and a specialized, efficient implementation is possible to obtain matchings of size around 0.866 of the maximum cardinality.

One-sided matching

The first matching heuristic we propose, OneSidedMatch, scales the given adjacency matrix $\mathbf{A}$ (each a_{ij} is originally either 0 or 1) and uses the scaled entries to randomly choose a column as a match for each row. The pseudocode of the heuristic is shown in Algorithm 6.

Algorithm 6: OneSidedMatch: Bipartite graphs

Data: $\mathbf{A}$: an $n \times n$, (0,1)-matrix with total support
Result: `cmatch`$[\cdot]$: the rows matched to columns

1 $\langle \mathbf{d}_r, \mathbf{d}_c \rangle \leftarrow$ ScaleSK($\mathbf{A}$)
2 **for** $j = 1$ *to* n **in parallel do**
3 `cmatch`$[j] \leftarrow$ *NIL*
4 **for** $i = 1$ **to** n **in parallel do**
5 Pick a random column $j \in \mathbf{A}_{i*}$ by using the probability density function

$$\frac{s_{ik}}{\Sigma_{\ell \in \mathbf{A}_{i*}} s_{i\ell}}, \text{for all } k \in \mathbf{A}_{i*}$$

where $s_{ik} = \mathbf{d}_r[i] \times \mathbf{d}_c[k]$ is the corresponding entry in the scaled matrix $\mathbf{S} = \mathbf{D}_R \mathbf{A} \mathbf{D}_C$
6 `cmatch`$[j] \leftarrow i$

OneSidedMatch first obtains the scaling vectors $\mathbf{d}_r$ and $\mathbf{d}_c$ corresponding to a doubly stochastic matrix $\mathbf{S}$ (line 1). After initializing the `cmatch` array, for each row i of $\mathbf{A}$, the heuristic randomly chooses a column $j \in \mathbf{A}_{i*}$ based on the probabilities computed by using corresponding scaled entries of row i. It then matches i and j. Clearly multiple rows can choose the same column and write to the same entry in `cmatch`. We assume that in the parallel, shared-memory setting, one of the write operation survives, and the `cmatch` array defines a valid matching, i.e., $\{\{$`cmatch`$[j], j\} : cmatch[j] \neq NIL\}$. We now analyze its approximation guarantee in terms of the matching cardinality.

Theorem 4.1. *Let* $\mathbf{A}$ *be an* $n \times n$*, (0,1)-matrix with total support. Then,* OneSidedMatch *obtains a matching of size at least* $n(1 - 1/e) \approx 0.632n$

in expectation as $n \to \infty$.

Proof. To compute the matching cardinality, we will count the columns that are not picked by any row and subtract it from n. Since $\Sigma_{k \in \mathbf{A}_{i*}} s_{ik} = 1$ for each row i of $\mathbf{S}$, the probability that a column j is not picked by any of the rows in $\mathbf{A}_{*j}$ is equal to $\prod_{i \in \mathbf{A}_{*j}} (1 - s_{ij})$. By applying the arithmetic-geometric mean inequality, we obtain

$$\sqrt[d_j]{\prod_{i \in \mathbf{A}_{*j}} (1 - s_{ij})} \leq \frac{d_j - \sum_{i \in \mathbf{A}_{*j}} s_{ij}}{d_j} \ ,$$

where $d_j = |\mathbf{A}_{*j}|$ is the degree of column vertex j. Therefore,

$$\prod_{i \in \mathbf{A}_{*j}} (1 - s_{ij}) \leq \left(1 - \frac{\sum_{i \in \mathbf{A}_{*j}} s_{ij}}{d_j}\right)^{d_j} \ .$$

Since $\mathbf{S}$ is doubly stochastic, we have $\sum_{i \in \mathbf{A}_{*j}} s_{ij} = 1$ and

$$\prod_{i \in \mathbf{A}_{*j}} (1 - s_{ij}) \leq \left(1 - \frac{1}{d_j}\right)^{d_j} \ .$$

The function on the right hand side above is an increasing one, and has the limit

$$\lim_{d_j \to \infty} \left(1 - \frac{1}{d_j}\right)^{d_j} = \frac{1}{e} \ ,$$

where e is the base of the natural logarithm. By the linearity of expectation, the expected number of unmatched columns is no larger than $\frac{n}{e}$. Hence, the cardinality of the matching is no smaller than $n\,(1 - 1/e)$. □

In Algorithm 6, we split the rows among the threads with a **parallel for** construct. For each row i, the corresponding thread chooses a random number r from a uniform distribution with range $(0, \sum_{k \in \mathbf{A}_{i*}} s_{ik}]$. Then, the last nonzero column index j for which $\sum_{1 \leq k \leq j} s_{ik} \leq r$ is found and `cmatch`$[j]$ is set to i. Since no synchronization or conflict detection is required, the heuristic promises significant speedups.

Two-sided matching

OneSidedMatch's approximation guarantee and suitable structure for parallel architectures make it a good heuristic. The natural question that follows is whether a heuristic with a better guarantee exits. Of course, the sought heuristic should also be simple and easy to parallelize. We asked: "what happens if we repeat the process for the other (column) side of the bipartite graph"? The question led us to the following algorithm. Let each

row select a column, and let each column select a row. Take all these $2n$ choices to construct a bipartite graph G (a subgraph of the input) with $2n$ vertices and at most $2n$ edges (if i chooses j and j chooses i, we have one edge), and seek a maximum cardinality matching in G. Since the number of edges is at most $2n$, any exact matching algorithm on this graph would be fast—in particular the worst case run time would be $\mathcal{O}(n^{1.5})$ [117]. Yet, we can do better and obtain a maximum cardinality matching in linear time by running the Karp-Sipser heuristic on G, as we display in Algorithm 7.

Algorithm 7: TwoSidedMatch: Bipartite graphs

Data: $\mathbf{A}$: an $n \times n$, $\{0,1\}$-matrix with total support
Result: `cmatch`$[\cdot]$: the rows matched to columns

1 $\langle \mathbf{d}_r, \mathbf{d}_c \rangle \leftarrow$ ScaleSK($\mathbf{A}$)
2 **for** $i = 1$ **to** n **in parallel do**
3 Pick a random column $j \in \mathbf{A}_{i*}$ by using the probability density function

$$\frac{s_{ik}}{\Sigma_{\ell \in \mathbf{A}_{i*}} s_{i\ell}}, \text{ for all } k \in \mathbf{A}_{i*}$$

where $s_{ik} = \mathbf{d}_r[i] \times \mathbf{d}_c[k]$ is the corresponding entry in the scaled matrix $\mathbf{S} = \mathbf{D}_R \mathbf{A} \mathbf{D}_C$.
4 `rchoice`$[i] \leftarrow j$
5 **for** $j = 1$ **to** n **in parallel do**
6 Pick a random row $i \in \mathbf{A}_{*j}$ by using the probability density function

$$\frac{s_{kj}}{\Sigma_{\ell \in \mathbf{A}_{*j}} s_{\ell j}}, \text{ for all } k \in \mathbf{A}_{*j}.$$

7 `cchoice`$[j] \leftarrow i$
8 Construct a bipartite graph $G = (V_R \cup V_C, E)$ where

$$E = \{\{i, \texttt{rchoice}[i]\} : i \in \{1, \ldots, n\}\} \cup \{\{\texttt{cchoice}[j], j\} : j \in \{1, \ldots, n\}\}.$$

9 `match` $\leftarrow$Karp-Sipser (G)

The most interesting component of TwoSidedMatch is the incorporation of the Karp-Sipser heuristic for two reasons. First, although it is only a heuristic, Karp-Sipser computes a maximum cardinality matching on the bipartite graph G constructed in Algorithm 7. Second, although Karp-Sipser has a sequential nature, we can obtain good speedups with a specialized parallel implementation. In general, it is hard to efficiently parallelize (non-trivial) graph algorithms, and it is even harder when the overall cost is $\mathcal{O}(n)$, which is the case for Karp-Sipser on G. We give a series of lemmas below which enables us to use Karp-Sipser as an exact algorithm with a good shared-memory parallel performance.

The first lemma describes the structure of G constructed at line 8 of

TwoSidedMatch.

Lemma 4.1. *Each connected component of G constructed in Algorithm 7 contains at most one simple cycle.*

Proof. A connected component $M \subseteq G$ with n' vertices can have at most n' edges. Let T be a spanning tree of M. Since T contains $n' - 1$ edges, the remaining edge in M can create at most one cycle when added to T. □

Lemma 4.1 explains why Karp-Sipser is an exact algorithm on G. If a component does not contain a cycle, Karp-Sipser consumes all its vertices in Phase 1. Therefore, all of the matching decisions given by Karp-Sipser are optimal for this component. Assume a component contains a simple cycle. After Phase 1, the component is either consumed, or due to Lemma 4.1, it is reduced to a simple cycle. In the former case, the matching is a maximum cardinality one. In the latter case, an arbitrary edge of the cycle can be used to match a pair of vertices. This decision necessarily leads to a unique perfect matching in the remaining simple path. These two arguments can be repeated for all the connected components to see that the Karp-Sipser heuristic finds a maximum cardinality matching in G.

Algorithm 8 describes our parallel implementation Karp-Sipser-MT. The graph is represented using a single array `choice`, where `choice`$[u]$ is the vertex randomly chosen by $u \in V_R \cup V_C$. The `choice` array is a concatenation of the arrays `rchoice` and `cchoice` set in TwoSidedMatch. Hence, an explicit graph construction for G (line 8 of Algorithm 7) is not required, and a transformation of the selected edges to a graph storage scheme is avoided. Karp-Sipser-MT uses three atomic operations for synchronization. The first operation _Add(*memory, value*) atomically adds a *value* to a *memory* location. It is used to compute the vertex degrees in the initial graph (line 9). The second operation _CompAndSwap(*memory, value, replace*) first checks whether the *memory* location has the *value*. If so, its content is *replace*d. The final content is returned. The third operation _AddAndFetch(*memory, value*) atomically adds a given *value* to a *memory* location and the final content is returned. We will describe the use of these two operations later.

Karp-Sipser-MT has two phases which correspond to the two phases of Karp-Sipser. The first phase of Karp-Sipser-MT is similar to that of Karp-Sipser in that optimal matching decisions are made about some degree-one vertices. The second phase of Karp-Sipser-MT handles remaining vertices very efficiently, without bothering with their degrees. The following definitions are used to clarify the difference between an original Karp-Sipser and Karp-Sipser-MT.

Definition 4.1. Given a matching and the array `choice`, let u be an unmatched vertex and $v =$ `choice`$[u]$. Then u is called:

Algorithm 8: Karp-Sipser-MT: Bipartite 1-out graphs

Data: $G = \{V, \texttt{choice}[\cdot]$: the chosen vertex for each $u \in V\}$
Result: $\texttt{match}[\cdot]$: the match array for $u \in V$

```
1  for all u ∈ V in parallel do
2      mark[u] ← 1
3      deg[u] ← 1
4      match[u] ← NIL
5  for all u ∈ V in parallel do
6      v ← choice[u]
7      mark[v] ← 0
8      if choice[v] ≠ u then
9          _Add(deg[v], 1)
10 for each vertex u in parallel do   /* Phase 1:  out-one vertices */
11     if mark[u] = 1 then
12         curr ← u
13         while curr ≠ NIL do
14             nbr ← choice[curr]
15             if _CompAndSwap(match[nbr], NIL, curr) = curr then
16                 match[curr] ← nbr
17                 curr ← NIL
18                 next ← choice[nbr]
19                 if match[next] = NIL then
20                     if _AddAndFetch(deg[next], −1) = 1 then
21                         curr ← next
22                     else
23                         curr ← NIL
24 for each column vertex u in parallel do   /* Phase 2:  the rest */
25     v ← choice[u]
26     if match[u] = NIL and match[v] = NIL then
27         match[u] ← v
28         match[v] ← u
```

- *out-one*, if v is unmatched, and no unmatched vertex w with $\texttt{choice}[w] = u$ exists.
- *in-one*, if v is matched, and only a single unmatched vertex w with $\texttt{choice}[w] = u$ exists.

The first phase of Karp-Sipser-MT (**for** loop of line 10) does not track and match all degree-one vertices. Instead, only the out-one vertices are taken into account. For each such vertex u that is already out-one before Phase 1, we have $\texttt{mark}[u] = 1$. Karp-Sipser-MT visits these vertices (lines 10-11). Newly arising out-one vertices are consumed right away without maintaining a list. The second phase of Karp-Sipser-MT (**for** loop of line 24) is much simpler than that of Karp-Sipser as the degrees of the vertices are not tracked/updated. We now discuss how these simplifications are possible while ensuring a maximum cardinality matching in G.

Observation 4.1. *An out-one or an in-one vertex is a degree-one vertex in Karp-Sipser.*

Observation 4.2. *A degree-one vertex in Karp-Sipser is either an out-one or an in-one vertex, or it is one of the two vertices u and v in a 2-clique such that $v = \texttt{choice}[u]$ and $u = \texttt{choice}[v]$.*

Lemma 4.2 (Proved elsewhere [76, Lemma 3])**.** *If there exists an in-one vertex in G at any time during the execution of Karp-Sipser-MT, an out-one vertex also exists.*

According to Observation 4.1, all the matching decisions given by Karp-Sipser-MT in Phase 1 are optimal, since an out-one vertex is a degree-one vertex. Observation 4.2 implies that among all the degree-one vertices, Karp-Sipser-MT ignores only the in-ones and 2-cliques. According to Lemma 4.2, an in-one vertex cannot exist without an out-one vertex, therefore they are handled in the same phase. The 2-cliques that survive Phase 1 are handled in Karp-Sipser-MT's Phase 2, since they can be considered as cycles.

To analyze the second phase of Karp-Sipser-MT, we will use the following lemma.

Lemma 4.3 (Proved elsewhere [76, Lemma 4])**.** *Let $G' = (V'_R \cup V'_C, E')$ be the graph induced by the remaining vertices after the first phase of Karp-Sipser-MT. Then, the set*

$$\{(u, \texttt{choice}[u]) : u \in V'_R, \texttt{choice}[u] \in V'_C\}$$

is a maximum cardinality matching in G'.

In the light of Observations 4.1 and 4.2 and Lemmas 4.1–4.3, Karp-Sipser-MT is an exact algorithm on the graphs created in Algorithm 7. Its worst case (sequential) run time is linear.

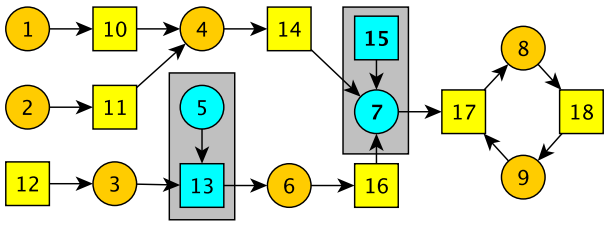


Figure 4.1: A toy bipartite graph with 9 row (circles) and 9 column (squares) vertices. The edges are oriented from a vertex u to the vertex $\texttt{choice}[u]$. Assuming all the vertices are currently unmatched, matching 15-7 (or 5-13) creates two degree-one vertices. But no out-one vertex arises after matching (15-7) and only one, vertex 6, arises after matching (5-13).

Karp-Sipser-MT tracks and consumes only the out-one vertices. This brings high flexibility while executing Karp-Sipser-MT in multi-threaded environments. Consider the example in Figure 4.1. Here, after matching a pair of vertices and removing them from the graph, multiple degree-one vertices can be generated. The standard Karp-Sipser uses a list to store these new degree-one vertices. Such a list is necessary to obtain a larger matching, but the associated synchronizations while updating it in parallel will be an obstacle for efficiency. The synchronization can be avoided up to some level if one sacrifices the approximation quality by not making all optimal decisions (as in some existing work [16]). We continue with the following lemma to take advantage of the special structure of the graphs in TwoSidedMatch for parallel efficiency in Phase 1.

Lemma 4.4 (Proved elsewhere [76, Lemma 5])**.** *Consuming an out-one vertex creates at most one new out-one vertex.*

According to Lemma 4.4, Karp-Sipser-MT does not need a list to store the new out-one vertices, since the process can continue with the new out-one vertex. In a shared-memory setting, there are two concerns for the first phase from the synchronization point of view. First, multiple threads that are consuming different out-one vertices can try to match them with the same unmatched vertex. To handle such cases, Karp-Sipser-MT uses the atomic _CompAndSwap operation (line 15 of Algorithm 8) and ensures that only one of these matchings will be processed. In this case, other threads, whose matching decisions are not performed, continue with the next vertex in the **for** loop at line 10. The second concern is that while consuming out-one vertices, several threads may create the same out-one vertex (and want to continue with it). For example, in Figure 4.1, when two threads consume the out-one vertices 1 and 2 at the same time, they both will try to continue with vertex 4. To handle such cases, an atomic _AddAndFetch

operation (line 20 of Algorithm 8) is used to synchronize the degree reduction operations on the potential out-one vertices. This approach explicitly orders the vertex consumptions and guarantees that only the thread who performs the last consumption before a new out-one vertex u appears continues with u. The other threads which wanted to continue with the same path stop and skip to the next unconsumed out-one vertex in the main **for** loop. One last concern that can be important on the parallel performance is the maximum length of such paths since a very long path can yield a significant imbalance on the work distribution to the threads. We investigated the length of such paths experimentally (see the original paper [76]) and observed them to be too short to hurt the parallel performance.

The second phase of Karp-Sipser-MT is efficiently parallelized by using the idea in Lemma 4.3. That is, a maximum cardinality matching for the graph remaining after the first phase of Karp-Sipser-MT can be obtained via a simple parallel **for** construct (see line 24 of Algorithm 8).

Quality of approximation. If the initial matrix $\mathbf{A}$ is the $n \times n$ matrix of 1s; that is $a_{ij} = 1$ for all $1 \leq i, j \leq n$, then the doubly stochastic matrix $\mathbf{S}$ is such that $s_{ij} = \frac{1}{n}$ for all $1 \leq i, j \leq n$. In this case, the graph G created by Algorithm 7 is a random 1-out bipartite graph [211]. Referring to a study by Meir and Moon [172], Karoński and Pittel [125] argue that the maximum cardinality of a matching in a random 1-out bipartite graph is $2(1-\Omega)n \approx 0.866n$ in expectation where $\Omega \approx 0.567$, also called Lambert's $W(1)$, is the unique solution of the equation $\Omega e^{\Omega} = 1$. We obtain the same result for random 1-out subgraph of any bipartite graph whose adjacency matrix has total support, as highlighted in the following theorem.

Theorem 4.2. *Let* $\mathbf{A}$ *be the* $n \times n$ *adjacency matrix of a given bipartite graph, where* $\mathbf{A}$ *has total support. Then,* TwoSidedMatch *obtains a matching of size* $2(1-\Omega)n \approx 0.866n$ *in expectation as* $n \to \infty$*, where* $\Omega \approx 0.567$ *is the unique solution of the equation* $\Omega e^{\Omega} = 1$.

Theorem 4.2 contributes to the known results about the Karp-Sipser heuristic (recall that it is known to leave out $\tilde{O}(n^{1/5})$ vertices) by showing a constant approximation ratio with some preprocessing. The existence of total support does not seem to be necessary for the Theorem 4.2 to hold (see the next subsection).

The proof of Theorem 4.2 is involved and can be found in the original paper [76]. It essentially adapts the original analysis of Karp and Sipser [128] to the special graphs at hand.

Discussion

The Sinkhorn-Knopp scaling algorithm converges linearly (when $\mathbf{A}$ has total support) where the convergence rate is equivalent to the square of the second

largest singular value of the resulting, doubly stochastic matrix [139]. If the matrix does not have support or total support, less is known about the Sinkhorn-Knopp scaling algorithm's convergence—we do not know how much of a reduction we will get at each step. In such cases, we are not able to bound the run time of the scaling step. However, the scaling algorithms should be run only a few iterations (see also the next paragraph) in practice, in which case the practical run time of our heuristics would be linear (in edges and vertices).

We have discussed the proposed matching heuristics while assuming that $\mathbf{A}$ has total support. This can be relaxed in two ways to render the overall approach practical in any bipartite graph. The first relaxation is that we do not need to run the scaling algorithms until convergence (as in some other use cases of similar algorithms [141]). If $\sum_{i \in \mathbf{A}_{*j}} s_{ij} \geq \alpha$ instead of $\sum_{i \in \mathbf{A}_{*j}} s_{ij} = 1$ for all $j \in \{1, \ldots, n\}$ then, $\lim_{d \to \infty} \left(1 - \frac{\alpha}{d}\right)^d = \frac{1}{e^\alpha}$. In other words, if we apply the scaling algorithms a few iterations, or until some relatively large error tolerance, we can still derive similar results. For example, if $\alpha = 0.92$, we will have a matching of size $n\left(1 - \frac{1}{e^\alpha}\right) \approx 0.601n$ (larger column sums give improved ratios; but there are columns whose sum is less than one, when the convergence is not achieved) for OneSidedMatch. With the same α, TwoSidedMatch will obtain an approximation of 0.840. In our experiments, the number of iterations were always a few, where the proven approximation guarantees were always observed. The second relaxation is that we do not need total support; we do not even need support nor equal number of vertices in the two vertex classes. Since little is known about the scaling methods on such matrices, we only mention some facts and observations, and later on, present some experiments to demonstrate the practicality of the proposed OneSidedMatch and TwoSidedMatch heuristics.

A sparse matrix (not necessarily square) can be permuted into a block upper triangular form using the canonical Dulmage-Mendelsohn (DM) decomposition [77]

$$\mathbf{A} = \begin{pmatrix} H & * & * \\ O & S & * \\ O & O & V \end{pmatrix}, \quad S = \begin{pmatrix} S_1 & * \\ O & S_2 \end{pmatrix}$$

where, H (horizontal) has more columns than rows and has a matching covering all rows; S is square and has a perfect matching; and V (vertical) has more rows than columns and a matching covering all columns. The following facts about the DM decomposition are well known [192, 194]. Any of these three blocks can be void. If H is not connected, then it is block diagonal with horizontal blocks. If V is not connected, then it is block diagonal with vertical blocks. If S does not have total support, then it is in block upper triangular form, shown on the right, where S_1 and S_2

have the same structure recursively, until each block S_i has total support. The entries in the blocks shown by "*" cannot be put into a maximum cardinality matching. When the presented scaling methods are applied to a matrix, the entries in "*" blocks will tend to zero (the case of S is well documented [202]). Furthermore, the row sums of the blocks of H will be a multiple of the column sums in the same block; a similar statement holds for V; finally S will be doubly stochastic. That is, the scaling algorithms applied to bipartite graphs without perfect matchings will zero out the entries in the irrelevant parts and identify the entries that can be put into a maximum cardinality matching.

4.2.3 Experiments

We present a selected set of results from the original paper [76]. We used a shared-memory parallel version of Sinkhorn-Knopp algorithm with a fixed number of iterations. This way, the scaling algorithm is simplified, as no convergence check is required. Furthermore, its overhead is bounded. In most of the experiments, we use $0, 1, 5$, or 10 scaling iterations, where the case of 0 corresponds to applying the matching heuristics with uniform edge selection probabilities. While these do not guarantee convergence of the Sinkhorn-Knopp scaling algorithm, it seems to be enough for our purposes.

Matching quality

We investigate the matching quality of the proposed heuristics on all square matrices with support from the SuiteSparse Matrix Collection [59] having at least 1000 non-empty rows/columns and at most 20000000 nonzeros (some of the matrices are also in the 10th DIMACS challenge [19]). There were 742 matrices satisfying these properties at the time of experimentation. With the OneSidedMatch heuristic, the quality guarantee of 0.632 was surpassed with 10 iterations of the scaling method in 729 matrices. With the TwoSidedMatch heuristic and the same number of iterations of the scaling methods, the quality guarantee of 0.866 was surpassed in 705 matrices. Making 10 more scaling iterations smoothed out the remaining instances. The Greedy heuristic (the second variant discussed in Section 4.2.1) obtained, on average (both the geometric and arithmetic means), matchings of size 0.93 of the maximum cardinality. In the worst case, 0.82 was observed.

We collect a few of the matrices from the SuiteSparse collection in Table 4.1. Figures 4.2a and 4.2b show the qualities on our test matrices. In the figures, the first columns represent the case that the neighbors are picked from a uniform distribution over the adjacency lists, i.e., the case with no scaling, hence no guarantee. The quality guarantees are achieved with only 5 scaling iterations. Even with a single iteration, the quality of TwoSidedMatch is more than 0.866 for all matrices, and that of OneSidedMatch

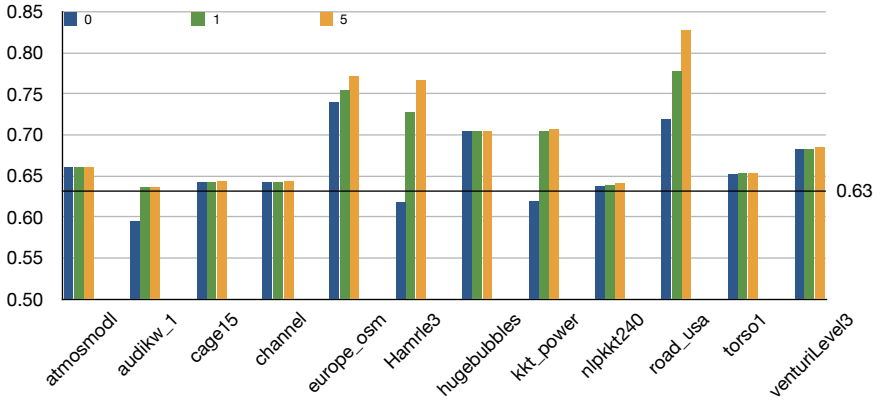
Name	n	# of edges	Avg. deg.	std_A	std_B	$\frac{sprank}{n}$
atmosmodl	1,489,752	10,319,760	6.9	0.3	0.3	1.00
audikw_1	943,695	77,651,847	82.2	42.5	42.5	1.00
cage15	5,154,859	99,199,551	19.2	5.7	5.7	1.00
channel	4,802,000	85,362,744	17.8	1.0	1.0	1.00
europe_osm	50,912,018	108,109,320	2.1	0.5	0.5	0.99
Hamrle3	1,447,360	5,514,242	3.8	1.6	2.1	1.00
hugebubbles	21,198,119	63,580,358	3.0	0.03	0.03	1.00
kkt_power	2,063,494	12,771,361	6.2	7.45	7.45	1.00
nlpkkt240	27,993,600	760,648,352	26.7	2.22	2.22	1.00
road_usa	23,947,347	57,708,624	2.4	0.93	0.93	0.95
torso1	116,158	8,516,500	73.3	419.59	245.44	1.00
venturiLevel3	4,026,819	16,108,474	4.0	0.12	0.12	1.00

Table 4.1: Properties of the matrices used in the comparisons. In the table, sprank is the maximum matching cardinality, Avg. deg. is the average vertex degree, std_A and std_B are the standard deviations of A and B vertex (row and column) degrees, respectively.

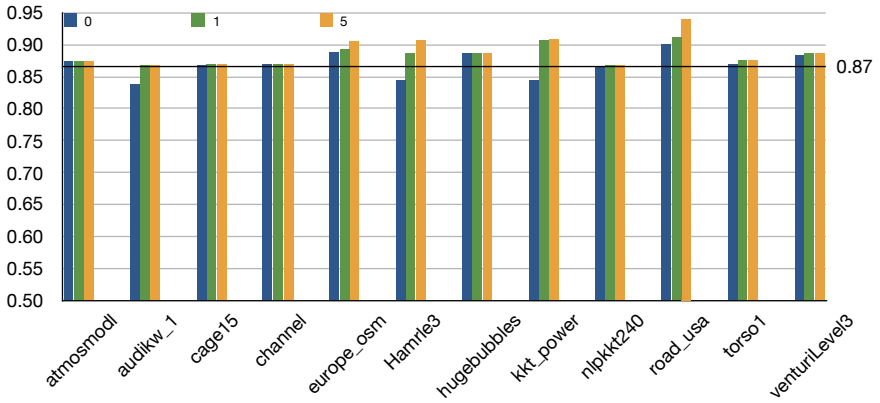
is more than 0.632 for all matrices.

Comparison of TwoSidedMatch with Karp-Sipser. Next, we analyze the performance of the proposed heuristics with respect to Karp-Sipser on a matrix class which we designed as a challenging case for Karp-Sipser. Let $\mathbf{A}$ be an $n \times n$ matrix, R_1 be the set of $\mathbf{A}$'s first $n/2$ rows, and R_2 be the set of $\mathbf{A}$'s last $n/2$ rows; similarly let C_1 and C_2 be the set of first $n/2$ and the set of last $n/2$ columns. As Figure 4.3 shows, $\mathbf{A}$ has a full $R_1 \times C_1$ block and an empty $R_2 \times C_2$ block. The last $h \ll n$ rows and columns of R_1 and C_1, respectively, are full. Each of the blocks $R_1 \times C_2$ and $R_2 \times C_1$ has a nonzero diagonal. Those diagonals form a perfect matching when combined. In the sequel, a matrix whose corresponding bipartite graph has a perfect matching will be called *full-sprank*, and *sprank-deficient* otherwise.

When $h \leq 1$, the Karp-Sipser heuristic consumes the whole graph during Phase 1 and finds a maximum cardinality matching. When $h > 1$, Phase 1 immediately ends, since there is no degree-one vertex. In Phase 2, the first edge (nonzero) consumed by Karp-Sipser is selected from a uniform distribution over the nonzeros whose corresponding rows and columns are still unmatched. Since the block $R_1 \times C_1$ is full, it is more likely that the nonzero will be chosen from this block. Thus, a row in R_1 will be matched with a column in C_1, which is a bad decision since the block $R_2 \times C_2$ is completely empty. Hence, we expect a decrease in the performance of Karp-Sipser as h increases. On the other hand the probability that TwoSidedMatch chooses an edge from that block goes to zero, as those entries cannot be in a perfect matching.



(a) OneSidedMatch



(b) TwoSidedMatch

Figure 4.2: Matching qualities of OneSidedMatch and TwoSidedMatch. The horizontal lines are at 0.632 and 0.866, respectively, which are the approximation guarantees for the heuristics. The legends contain the number of scaling iterations.

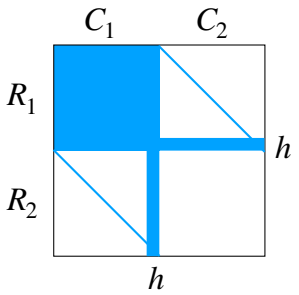


Figure 4.3: Adjacency matrices of hard bipartite graph instances for Karp-Sipser.

		Results with different number of scaling iterations						
	Karp-	0	1		5		10	
h	Sipser	Qual.	Err.	Qual.	Err.	Qual.	Err.	Qual.
2	0.782	0.522	13.853	0.557	3.463	0.989	0.578	0.999
4	0.704	0.489	11.257	0.516	3.856	0.980	0.604	0.997
8	0.707	0.466	8.653	0.487	4.345	0.946	0.648	0.996
16	0.685	0.448	6.373	0.458	4.683	0.885	0.725	0.990
32	0.670	0.447	4.555	0.453	4.428	0.748	0.867	0.980

Table 4.2: Quality comparison (minimum of 10 executions for each instance) of the Karp-Sipser heuristic and TwoSidedMatch on matrices described in Fig. 4.3 with $n = 3200$ and $h \in \{2, 4, 8, 16, 32\}$.

The results of the experiments are in Table 4.2. The first column shows the h value. Then the *matching quality* obtained by Karp-Sipser, and by TwoSidedMatch with different number of scaling iterations (0, 1, 5, 10), as well as the scaling error are given. The scaling error is the maximum difference between 1 and each row/column sum of the scaled matrix (for 0 iterations it is equal to $n - 1$ for all cases). The quality of a matching is computed by dividing its cardinality to the maximum one, which is $n = 3200$ for these experiments. To obtain the values in each cell of the table, we run the programs 10 times and give the minimum quality (as we are investigating the worst-case behavior). The highest variance for Karp-Sipser and TwoSidedMatch were (up to four significant digits) 0.0041 and 0.0001, respectively. As expected, when h increases, Karp-Sipser performs worse, and the matching quality drops to 0.67 for $h = 32$. TwoSidedMatch's performance increases with the number of scaling iterations. As the experiment shows, only 5 scaling iterations are sufficient to make the proposed two-sided matching heuristic significantly better than Karp-Sipser. However, this number was not enough to reach 0.866 for the matrix with $h = 32$. On this matrix, with 10 iterations, only 2% of the rows/columns remain unmatched.

Matching quality on bipartite graphs without perfect matchings. We analyze the proposed heuristics on a class of random sprank-deficient square ($n = 100000$) and rectangular ($m = 100000$ and $n = 120000$) matrices with a uniform nonzero distribution (two more sprank-deficient matrices are used in the scalability tests as well). These matrices are generated by Matlab's `sprand` command (generating Erdös-Rényi random matrices [85]). The total number of nonzeros is set to be around $d \times 100000$ for $d \in \{2, 3, 4, 5\}$. The top half of Table 4.3 presents the results of this experiment with square matrices, and the bottom half presents the results with the rectangular ones.

As in the previous experiments, the matching qualities in the table are the minimum of 10 executions for the corresponding instances. As Table 4.3

d	iter	sprank	One Sided Match	Two Sided Match	d	iter	sprank	One Sided Match	Two Sided Match
Square 100000 × 100000									
2	0	78,225	0.770	0.912	4	0	97,787	0.644	0.838
2	1	78,225	0.797	0.917	4	1	97,787	0.673	0.848
2	5	78,225	0.850	0.939	4	5	97,787	0.719	0.873
2	10	782,25	0.879	0.954	4	10	97,787	0.740	0.886
3	0	92,786	0.673	0.851	5	0	99,223	0.635	0.840
3	1	92,786	0.703	0.857	5	1	99,223	0.662	0.851
3	5	92,786	0.756	0.884	5	5	99,223	0.701	0.873
3	10	92,786	0.784	0.902	5	10	99,223	0.716	0.882
Rectangular 100000 × 120000									
2	0	87,373	0.793	0.912	4	0	99,115	0.729	0.899
2	1	87,373	0.815	0.918	4	1	99,115	0.754	0.910
2	5	87,373	0.861	0.939	4	5	99,115	0.792	0.933
2	10	87,373	0.886	0.955	4	10	99,115	0.811	0.946
3	0	96,564	0.739	0.896	5	0	99,761	0.725	0.905
3	1	96,564	0.769	0.904	5	1	99,761	0.749	0.917
3	5	96,564	0.813	0.930	5	5	99,761	0.781	0.936
3	10	96,564	0.836	0.945	5	10	99,761	0.792	0.943

Table 4.3: Matching qualities of the proposed heuristics on random sparse matrices with uniform nonzero distribution. Square matrices in the top half, rectangular matrices in the bottom half. d: average number of nonzeros per row.

shows, when the deficiency is high (correlated with small d), it is easier for our algorithms to approximate the maximum cardinality. However, when d gets larger, the algorithms require more scaling iterations. Even in this case, 5 iterations are sufficient to achieve the guaranteed qualities. In the square case, the minimum quality achieved by OneSidedMatch and TwoSidedMatch were 0.701 and 0.873 In the rectangular case, the minimum quality achieved by OneSidedMatch and TwoSidedMatch were 0.781 and 0.930, respectively, with 5 scaling iterations. In all cases, increased scaling iterations results in higher quality matchings, in accordance with the previous results on square matrices with perfect matchings.

Comparisons with some other codes

We run Azad et al.'s [16] variant of Karp-Sipser (`ksV`), the two greedy matching heuristics of Blelloch et al. [29] (`inc` and `nd` which are referred to as "incremental" and "non-deterministic reservation" in the original paper), and the proposed OneSidedMatch and TwoSidedMatch heuristics with one and 16 threads on the matrices given in Table 4.1. We also add one more matrix `wc_50K_64` from the family shown in Fig. 4.3 with $n = 50000$ and $h = 64$.

	Run time in seconds									
	One thread					16 threads				
Name	ksV	inc	nd	1SD	2SD	ksV	inc	nd	1SD	2SD
atmosmodl	0.20	14.80	0.07	0.12	0.30	0.03	1.63	0.01	0.01	0.03
audikw_1	0.98	206.00	0.22	0.55	0.82	0.10	16.90	0.02	0.06	0.09
cage15	1.54	9.96	0.39	0.92	1.95	0.29	1.10	0.04	0.09	0.19
channel	1.29	105.00	0.30	0.82	1.46	0.14	9.07	0.03	0.08	0.13
europe_osm	12.03	51.80	2.06	4.89	11.97	1.20	7.12	0.21	0.46	1.05
Hamrle3	0.15	0.12	0.04	0.09	0.25	0.02	0.02	0.01	0.01	0.02
hugebubbles	8.36	4.65	1.28	3.92	10.04	0.95	0.53	0.18	0.37	0.94
kkt_power	0.55	0.68	0.09	0.19	0.45	0.08	0.16	0.01	0.02	0.05
nlpkkt240	10.88	1900.00	2.56	5.56	10.11	1.15	237.00	0.23	0.58	1.06
road_usa	6.23	3.57	0.96	2.16	5.42	0.70	0.38	0.09	0.22	0.50
torso1	0.11	7.28	0.02	0.06	0.09	0.02	1.20	0.00	0.01	0.01
venturiLevel3	0.45	2.72	0.13	0.32	0.84	0.08	0.29	0.01	0.04	0.09
wc_50K_64	7.21	3200.00	1.46	4.39	5.80	1.17	402.00	0.12	0.55	0.59
	Quality of the obtained matching									
Name	ksV	inc	nd	1SD	2SD	ksV	inc	nd	1SD	2SD
atmosmodl	1.00	1.00	1.00	0.66	0.87	0.98	1.00	0.97	0.66	0.87
audikw_1	1.00	1.00	1.00	0.64	0.87	0.95	1.00	0.97	0.64	0.87
cage15	1.00	1.00	1.00	0.64	0.87	0.95	1.00	0.91	0.64	0.87
channel	1.00	1.00	1.00	0.64	0.87	0.99	1.00	0.99	0.64	0.87
europe_osm	0.98	0.95	0.95	0.75	0.89	0.98	0.95	0.94	0.75	0.89
Hamrle3	1.00	0.84	0.84	0.75	0.90	0.97	0.84	0.86	0.75	0.90
hugebubbles	0.99	0.96	0.96	0.70	0.89	0.96	0.96	0.90	0.70	0.89
kkt_power	0.85	0.79	0.79	0.78	0.89	0.87	0.79	0.86	0.78	0.89
nlpkkt240	0.99	0.99	0.99	0.64	0.86	0.99	0.99	0.99	0.64	0.86
road_usa	0.94	0.81	0.81	0.76	0.88	0.94	0.81	0.81	0.76	0.88
torso1	1.00	1.00	1.00	0.65	0.88	0.98	1.00	0.98	0.65	0.87
venturiLevel3	1.00	1.00	1.00	0.68	0.88	0.97	1.00	0.96	0.68	0.88
wc_50K_64	0.50	0.50	0.50	0.81	0.98	0.70	0.50	0.51	0.81	0.98

Table 4.4: Run time and quality of different heuristics. OneSidedMatch and TwoSidedMatch are labeled with 1SD and 2SD, respectively, and apply two steps of scaling iterations. The heuristic `ksV` is from Azad et al. [16], and the heuristics `inc` and `nd` are from Blelloch et al. [29].

The OneSidedMatch and TwoSidedMatch heuristics are run with two scaling iterations, and the reported run time includes all components of the heuristics. The `inc` and `nd` heuristics are compiled with `g++ 4.4.5` and `ksV` is compiled with `gcc 4.4.5`. For all the heuristics, we used `-O2` optimization and the appropriate OpenMP flag for compilation. The experiments were carried out on a machine equipped with two Intel Sandybridge-EP CPUs clocked at 2.00Ghz and 256GB of memory split across the two NUMA domains. Each CPU has eight-cores (16 cores in total) and HyperThreading is enabled. Each core has its own 32kB L1 cache and 256kB L2 cache. The 8 cores on a CPU share a 20MB L3 cache. The machine runs 64-bit Debian with Linux 2.6.39-bpo.2-amd64. The run time of all heuristics (in seconds) and their matching qualities are given in Table 4.4.

As seen in Table 4.4, `nd` is the fastest of the tested heuristics with both one thread and 16 threads on this data set. The second fastest heuristic is OneSidedMatch. The heuristic `ksV` is faster than TwoSidedMatch on six instances with one thread. With 16 threads, TwoSidedMatch is almost always faster than `ksV`—both heuristics are quite efficient and have a maximum run time of 1.17 seconds. The heuristic `inc` was observed to have large run time on some matrices with relatively high nonzeros per row. All the existing heuristics are always very efficient, except `inc` which had difficulties on some matrices in the data set. In the original reference [29], `inc` is quite efficient for some other matrices (where the codes are compiled with Cilk). We do not see run time with `nd` elsewhere, but in our data set it was always better than `inc`. The proposed OneSidedMatch heuristic's quality is almost always close to its theoretical limit. TwoSidedMatch also obtains quality results close to its theoretical limit. Looking at the quality of the matching with one thread, we see that `ksV` obtains (almost always) the best score, except on `kkt_power` and `wc_50K_64`, where TwoSidedMatch obtains the best score. The heuristics `inc` and `nd` obtain the same score with one thread, but with 16 threads `inc` obtains better quality than `nd` almost always. OneSidedMatch never obtains the best score (TwoSidedMatch is always better), yet it is better than `ksV`, `inc` and `nd` on the synthetic `wc_50K_64` matrix. The TwoSidedMatch heuristic obtains better results than `inc` and `nd` on `Hamrle3`, `kkt_power`, `road_usa`, and `wc_50K_64` with both one thread and 16 threads. Also on `hugebubbles` with 16 threads, the difference between TwoSidedMatch and `nd` is 1% (in favor of `nd`).

4.3 Approximation algorithms for general undirected graphs

We extend the results on bipartite graphs of the previous section to general undirected graphs. We again select a subgraph randomly with a probability density function obtained by scaling the adjacency matrix of the input graph, and run Karp-Sipser [128] on the selected subgraph. Again, Karp-Sipser becomes an exact algorithm on the selected subgraph, and analysis reveals that the approximation ratio is around 0.866 of the maximum cardinality in the original graph. We also propose two other variants of this algorithm, which obtain better results both in theory and in practice. We omit most of the proofs in the presentation; they can be found in the associated paper [73].

Recall that a graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Let $G^k_{out} = (V, E')$ be a random subgraph of G where each vertex in V randomly chooses k of its edges, with repetition (an edge $\{u, v\}$ is included in G^k_{out} only once even if it is chosen multiple times). We call G^k_{out} as a k-out subgraph of G.

A permutation of $[1, \ldots, n]$ in which no element stays in its original po-

sition is called *derangement*. The total number of derangements of $[1, \ldots, n]$ is denoted by $!n$, and is the nearest integer to $\frac{n!}{e}$ [36, pp. 40–42], where e is the base of the natural logarithm.

4.3.1 Related work

The first polynomial time algorithm with $\mathcal{O}(n^2\tau)$ complexity for the maximum cardinality matching problem on general graphs with n vertices and τ edges is proposed by Edmonds [79]. Currently, the fastest algorithms have $\mathcal{O}(\sqrt{n}\tau)$ complexity [30, 92, 174]. The first of these algorithms is by Micali and Vazirani [174]. recently, its simpler analysis is given by Vazirani [210]. Later, Blum [30] and Gabow and Tarjan [92] proposed algorithms with the same complexity.

Random graphs have been frequently analyzed in terms of their maximum matching cardinality. Frieze [90] shows results for undirected (general) graphs along the lines of Walkup's results for bipartite graphs [211]. In particular, he shows that if each vertex chooses uniformly at random one neighbor (among all vertices), then the constructed graph does not have a perfect matching almost always. In other words, he shows that random 1-out subgraphs of a complete graph do not have perfect matchings. Luckily, as in bipartite graphs, if each vertex chooses two neighbors, then the constructed graph has perfect matchings almost always. That is, random 2-out subgraphs of a complete graph have perfect matchings. Our contributions are about making these statements valid for any host graph (assuming the adjacency matrix has total support), and measure the approximation guarantee of the 1-out case.

Randomized algorithms which check the existence of a perfect matching and generate perfect/maximum matchings have been proposed in the literature [45, 120, 165, 177, 178]. Lovász showed that the existence of a perfect matching can be verified in randomized $\mathcal{O}(n^\omega)$ time where $\mathcal{O}(n^\omega)$ is the time complexity of the fastest matrix multiplication algorithm available [165]. More information such as the set of allowed edges, i.e., the edges in some maximum matching, of a graph can also be found with the same randomized complexity as shown by Cheriyan [45].

To construct a maximum cardinality matching in a general, non-bipartite graph, a simple, easy to implement algorithm with $\mathcal{O}(n^{\omega+1})$ randomized complexity is proposed by Rabin and Vazirani [197]. The algorithm uses matrix inversion as a subroutine. Later, Mucha and Sankowski proposed the first algorithm for the same problem with $\mathcal{O}(n^\omega)$ randomized complexity [178]. This algorithm uses expensive sub-routines such as Gaussian elimination and equivalence classes formed based on the edges that appear in some perfect matching [166]. A simpler randomized algorithm with the same complexity is proposed by Harvey [108].

4.3.2 One-Out, its analysis, and variants

We first present the main heuristic, and then analyze its approximation guarantee. While the heuristic is a straightforward adaptation of its counterpart for bipartite graphs [76], the analysis is more complicated, because of odd cycles. The analysis shows that random 1-out subgraphs of a given graph have the maximum cardinality of a matching around $0.866 - \log(n)/n$ of the best possible—the observed performance (see Section 4.3.3) is higher. Then two variants of the main heuristic are discussed without proofs. They extend the 1-out subgraphs obtained by the first one, and hence deliver better results theoretically.

The main heuristic One-Out

The heuristic shown in Algorithm 9 first scales the adjacency matrix of a given undirected graph to be doubly stochastic. Based on the values in the matrix, the heuristic then randomly marks one of the neighbors for each vertex as chosen. This defines one marked edge per vertex. Then, the subgraph of the original graph containing only the marked edges is formed, yielding an undirected graph with at most n edges, each vertex having at least one edge incident on it. The Karp-Sipser heuristic is run on this subgraph and finds a maximum cardinality matching on it, as the resulting 1-out graph has unicyclic components. The approximation guarantee of the proposed heuristic is analyzed for this step. One practical improvement to the previous work is to make sure that the matching obtained is a maximal one by running Karp-Sipser on the vertices that are not matched. This brings in a large improvement to the cardinality of the matching in practice.

Let us classify the edges incident on a vertex v_i as in-edges (from those neighbors that have chosen i at Line 4) and an out-edge (to a neighbor chosen by i at Line 4. Dufossé et al. [76, Lemma 3] show that during any execution of Karp-Sipser, it suffices to consider the vertices whose in-edges are unavailable but out-edges are available as degree-1 vertices.

The analysis traces an execution of Karp-Sipser on the subgraph G^1_{out}. In other words, the analysis can also be perceived as the analysis of the Karp-Sipser heuristic's performance on random 1-out graphs. Let A_1 be the set of vertices not chosen by any other vertex at Line 4 of Algorithm 9. These vertices have in-degree zero and out degree one, and hence can be processed by Karp-Sipser. Let B_1 be the set of vertices chosen by the vertices in A_1. The vertices in B_1 can be perfectly matched with vertices in A_1; leaving some A_1 vertices not matched and creating some new in-degree-0 vertices. We can proceed to define A_2 to be the vertices that have in degree-0 in $V \setminus (A_1 \cup B_1)$, and define B_2 as those chosen by A_2, and so on so forth. Formally, let B_0 be an empty set, and define A_i to be the set of vertices with in-degree 0 in $V \setminus B_{i-1}$, and B_i be the vertices chosen by those in A_i,

Algorithm 9: ONE-OUT: Undirected graphs

Data: $G = (V, E)$ and its adjacency matrix $\mathbf{A}$
Result: `match`[·]: the matching

1 $\mathbf{D} \leftarrow$ SYMSCALE($\mathbf{A}$)
2 **for** $i = 1$ **to** n **in parallel do**
3 Pick a random $j \in \mathbf{A}_{i*}$ by using the probability density function

$$\frac{s_{ik}}{\Sigma_{t \in \mathbf{A}_{i*}} s_{it}}, \text{for all } k \in \mathbf{A}_{i*}$$

where $s_{ik} = \mathbf{d}[i] \times \mathbf{d}[k]$ is the corresponding entry in the scaled matrix $\mathbf{S} = \mathbf{DAD}$.
4 Mark that i chooses j
5 Construct a graph $G^1_{out} = (V, E)$, where

$$V = \{1, \ldots, n\}$$
$$E = \{(i, j) : i \text{ chose } j \text{ or } j \text{ chose } i\}$$

6 `match` $\leftarrow$KARPSIPSER$_{\text{ONE-OUT}}(G^1_{out})$
7 Make `match` a maximal matching

for $i \geq 1$. Notice that $A_i \subseteq A_{i+1}$ and $B_i \subseteq B_{i+1}$. The Karp-Sipser heuristic can process A_1, then $A_2 \setminus A_1$, and so on, until the remaining graph has cycles only. The sets A_i and B_i and their cardinality are at the core of our analysis. We first present some facts about these sets and their cardinality, and describe an implementation of Karp-Sipser instrumented to highlight them.

Lemma 4.5. *With the definitions above, $A_i \cap B_i = \emptyset$.*

Proof. We prove this by induction. For $i = 1$ it clearly holds. Assume that it holds for all $i < \ell$. Suppose there exists a vertex $u \in A_\ell \cap B_\ell$. Because $A_{\ell-1} \cap B_{\ell-1} = \emptyset$, u must necessarily belong to both $A_\ell \setminus A_{\ell-1}$ and $B_\ell \setminus B_{\ell-1}$. For u to be in $B_\ell \setminus B_{\ell-1}$, there must exist at least one vertex $v \in A_\ell \setminus A_{\ell-1}$ such that v chooses u. However the condition for $u \in A_\ell$ is that no vertex in $V \cap (A_{\ell-1} \cup B_{\ell-1})$ has selected it. This is a contradiction and the intersection $A_\ell \cap B_\ell$ should be empty. □

Corollary 4.1. *$A_i \cap B_j = \emptyset$ for $i \leq j$.*

Proof. Assume $A_i \cap B_j \neq \emptyset$. Since $A_i \subseteq A_j$ we have a contradiction as $A_j \cap B_j = \emptyset$ by Lemma 4.5. □

Thanks to Lemma 4.5 and Corollary 4.1, the sets A_i and B_i are disjoint, and they form a bipartite subgraph of G^1_{out}, for all $i = 1, \ldots, \ell$.

The proposed version Karp-Sipser$_{\text{ONE-OUT}}$ of the Karp-Sipser heuristic for 1-out graphs is shown in Algorithm 10. The degree-1 vertices are kept

in a first-in first-out priority queue Q. The queue is first initialized with A_1, and the character '#' is used to mark the end of A_1. Then, all vertices in A_1 are matched to some other vertices, defining B_1. When we remove two matched vertices from the graph G^1_{out} at Lines 29 and 40, we update the degrees of their remaining neighbors, and append the vertices which have degrees of one to the queue. During Phase-1 of Karp-Sipser$_{\text{ONE-OUT}}$, we also maintain the set of A_i and B_i vertices, while storing only the last one. A_ℓ and B_ℓ are returned along with the number ℓ of levels, which is computed thanks to the use of the marker #.

Apart from the scaling step, the proposed heuristic in Algorithm 9 has linear worst-case time complexity of $O(n + \tau)$. The scaling step, if applied until convergence, can take more time than that. We do do not suggest running it until convergence; for all practical purposes 5 or 10 iterations of the basic method [141] or even less of the Newton iterations [140] seem sufficient (see experiments). Therefore, the practical run time of the algorithm is linear.

Analysis of ONE-OUT

Let a_i and b_i be two random variables representing the cardinalities of A_i and B_i, respectively, in an execution of Karp-Sipser$_{\text{ONE-OUT}}$ on a random 1-out graph. Then, Karp-Sipser$_{\text{ONE-OUT}}$ matches b_ℓ edges in the first phase, and leaves $a_\ell - b_\ell$ vertices unmatched. What remains after the first phase is a set of cycles. In the bipartite graph case [76], all vertices in those cycles are matchable and hence the cardinality of the matching was measured by $n - a_\ell + b_\ell$. Since we can possibly have odd cycles after the first phase, we cannot match all remaining vertices in the general case of undirected graphs. Let c be a random variable representing the number of odd cycles after the first phase of Karp-Sipser$_{\text{ONE-OUT}}$. Then we have the following (obvious) lemma about the approximation guarantee of Algorithm 9.

Lemma 4.6. *At the end of execution, the number of unmatched vertices is $a_\ell - b_\ell + c$. Hence, Algorithm 9 matches at least $n - (a_\ell - b_\ell + c)$ vertices.*

We need to quantify $a_\ell - b_\ell$ and c in Lemma 4.6. We state an an upper bound on $a_\ell - b_\ell$ in Lemma 4.7, and an upper bound on c in Lemma 4.8, without any proof (the reader can find the proofs in the original paper [73]). While the bound for $a_\ell - b_\ell$ holds for any graph with total support presented to Algorithm 9, the bounds for c are shown for random 1-out graphs of complete graphs. By plugging the bounds for these quantities, we obtain the following theorem on the approximation guarantee of Algorithm 9.

Theorem 4.3. *Algorithm 9 obtains a matching with cardinality at least $0.866 - \frac{\lceil 1.04 \log(0.336n) \rceil}{n}$ in expectation, when the input is a complete graph.*

Algorithm 10: Karp-Sipser$_{\text{ONE-OUT}}$: Undirected graphs

Data: $G^1_{out} = (V, E)$
Result: `match`$[\cdot]$: the mates of vertices
Result: ℓ: the number of levels in the first phase
Result: A: the set of degree-1 vertices in the first phase
Result: B: the set of vertices matched to A vertices

`match`$[u] \leftarrow$ NIL for all u
$Q \leftarrow \{v : \texttt{deg}(v) = 1\}$`/* degree-1 or in-degree 0 */`
if $Q = \emptyset$ **then**
  $\ell \leftarrow 0$`/* no vertex in level 1 */`
else
  $\ell \leftarrow 1$
ENQUEUE$(Q, \#)$`/* marks the end of the first level */`
Phase-1$\leftarrow$ ongoing
$A \leftarrow B \leftarrow \emptyset$
while true do
  while $Q \neq \emptyset$ **do**
    $u \leftarrow$ DEQUEUE(Q)
    if $u = \#$ **and** $Q = \emptyset$ **then**
      break the while-Q-loop
    else if $u = \#$ **then**
      $\ell \leftarrow \ell + 1$
      ENQUEUE$(Q, \#)$`/* new level formed */`
      skip to the next while-Q-iteration
    if `match`$[u] \neq$ *NIL* **then**
      skip to the next while-Q-iteration
    for $v \in \text{adj}(u)$ **do**
      if `match`$[v] =$ *NIL* **then**
        `match`$[u] \leftarrow v$
        `match`$[v] \leftarrow u$
        if *Phase-1 = ongoing* **then**
          $A \leftarrow A \cup \{u\}$
          $B \leftarrow B \cup \{v\}$
        $N \leftarrow \text{adj}(v)$
        $G^1_{out} \leftarrow G^1_{out} \setminus \{u, v\}$
        ENQUEUE(Q, w) for $w \in N$ and $\deg(w) = 1$
        break the for-v-loop
    if *Phase-1 = ongoing* **and** `match`$[u] =$ *NIL* **then**
      $A \leftarrow A \cup \{u\}$`/* u cannot be matched */`
  Phase-1 $\leftarrow$ done
  if $E \neq \emptyset$ **then**
    pick a random edge (u, v)
    `match`$[u] \leftarrow v$
    `match`$[v] \leftarrow u$
    $N \leftarrow \text{adj}(\{u, v\})$
    $G^1_{out} \leftarrow G^1_{out} \setminus \{u, v\}$
    ENQUEUE(Q, w) for $w \in N$ and $\deg(w) = 1$
  else
    break the while-true loop

The theorem can also be interpreted more theoretically as a random 1-out graph has a maximum cardinality of a matching at least $0.866 - \frac{\lceil 1.04 \log(0.336n) \rceil}{n}$, in expectation. The bound is in close vicinity of 0.866, which was the proved bound for the bipartite case [76]. We note that in deriving this bound we assumed a random 1-out graph (as opposed to a random 1-out subgraph of a given graph) only at a single step. We leave the extension to this latter case as future work and present experiments suggesting that the bound is also achieved for this case. In Section 4.3.3, we empirically show that the same bound also holds for graphs whose corresponding matrices do not have total support.

In order to measure $a_\ell - b_\ell$, we adapted a proof from earlier work [76], which was inspired by Karp and Sipser's analysis of the first phase of their heuristic [128] and obtained the following lemma.

Lemma 4.7. *$a_\ell - b_\ell \leq (2\Omega - 1)n$, where $\Omega \approx 0.567$ equals to $W(1)$ of Lambert's W function.*

The lemma also reads as $a_\ell - b_\ell \leq \sum_{i=1}^{n}(2 \cdot 0.567 - 1) \leq 0.134 \cdot n$.

In order to achieve the result of Theorem 4.3, we need to bound c, the number of odd cycles that remain on a G^1_{out} graph after the first phase of Karp-Sipser$_{\text{ONE-OUT}}$. We were able to bound c on random 1-out graphs (that is we did not show it for a general graph).

Lemma 4.8. *The expected number c of cycles remaining after the first phase of Karp-Sipser$_{\text{ONE-OUT}}$ in random 1-out graphs is less than or equal to $\lceil 1.04 \log(n - a_\ell - b_\ell) \rceil$, which is also less than or equal to $\lceil 1.04 \log(0.336n) \rceil$.*

Variants

Here we summarize two related theoretical random bipartite graph models that we adapt to the undirected case using similar algorithms. The presentation is brief and without proofs; we will present experiments in Section 4.3.3.

The random $(1 + e^{-1})$ undirected graph model (see the bipartite version [125] summarized in Section 4.2.1) lets each vertex choose a random neighbor. Then, the vertices that have not been chosen select one more neighbor. The maximum cardinality of a matching in the subgraph consisting of the identified edges can be computed as an approximate matching in the original graph. As this heuristic uses a richer graph structure than 1-out, we expect perfect or near perfect matchings in the general case.

A model richer in edges is the random 2-out graph model. In this model, each vertex chooses two neighbors. There are two enticing characteristics of this model in the uniform bipartite case. First, the probability of having a perfect matching goes to one with the increasing number of vertices [211]. Second, there is a special algorithm for computing the maximum cardinality matchings in these (bipartite) graphs [127] with high probability, in linear time in expectation.

4.3.3 Experiments

To understand the effectiveness and efficiency of the proposed heuristics in practice, we report the matching quality and various statistics regarding the sets that appear in our analyses in Section 4.3.2. For the experiments, we used three graph datasets. The first set is generated with matrices from the SuiteSparse Matrix Collection [59]. We investigated all $n \times n$ matrices from the collection with $50000 \leq n \leq 100000$. For a matrix from this set, we removed the diagonal entries, symmetrized the pattern of the resulting matrix, and discarded a matrix if it has empty rows. There were 115 matrices at the end which we used as the adjacency matrix. The graphs in the second dataset are synthetically generated to make Karp-Sipser deviate from the optimal matching as much as possible. This dataset contains five graphs with different hardness levels for Karp-Sipser. The third set contains five large, real-life matrices from the SuiteSparse collection for measuring the run time efficiency of the proposed heuristics.

A comprehensive evaluation

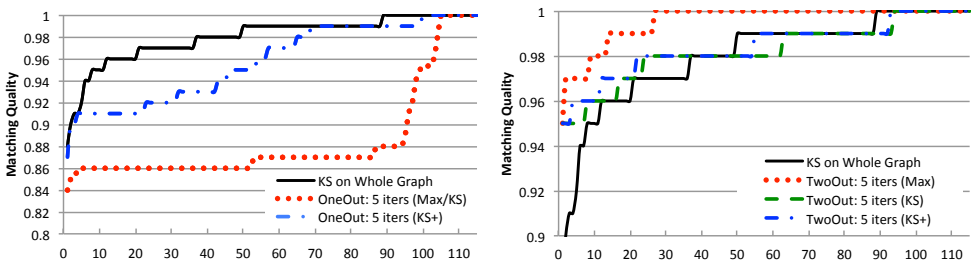
We use MATLAB to measure the matching qualities based on the first two datasets. For each matrix, five runs are performed with each randomized matching heuristic and the average is reported. One, five and ten iterations are performed to evaluate the impact of the scaling method.

Table 4.5 summarizes the quality of the matchings for all the experiments on the first dataset. The matching quality is measured as the ratio of the matching cardinality to the maximum cardinality matching in the original graph. The table presents statistics for matching qualities of Karp-Sipser performed on the original graphs (first row), 1-out graphs (the second set of rows), Karoński-Pittel-like $(1 + e^{-1})$-out graphs (the third set of rows), and 2-out graphs (the last set of rows).

For the U-Max rows, we construct k-out graphs by using uniform probabilities while selecting the neighbors as proposed in the literature [90, 125]. We compare the cardinality of the maximum matchings in these k-out graphs to the maximum matching cardinality on the original graphs and report the statistics. The rows St-Max report the same statistics for the k-out graphs constructed by using probabilities with $t \in \{1, 5, 10\}$ scaling iterations. These statistics serve as upper bounds on the matching qualities of the proposed St-KS heuristics which execute Karp-Sipser on the k-out graphs obtained with t scaling iterations. Since St-KS heuristics use only a subgraph, the matchings they obtain are not maximal with respect to the original edge set. The proposed St-KS+ heuristics exploit this fact and apply another Karp-Sipser phase on the subgraph containing only the unmatched vertices to improve the quality of the matchings. The table does not contain St-Max rows for 1-out graphs since Karp-Sipser is an optimal

	Alg.	Min	Max	Avg.	GMean	Med.	StDev
	Karp-Sipser	0.880	1.000	0.980	0.980	0.988	0.022
1-out	U-Max	0.168	1.000	0.846	0.837	0.858	0.091
	S1-KS	0.479	1.000	0.869	0.866	0.863	0.059
	S5-KS	0.839	1.000	0.885	0.884	0.865	0.044
	S10-KS	0.858	1.000	0.889	0.888	0.866	0.045
	S1-KS+	0.836	1.000	0.951	0.950	0.953	0.043
	S5-KS+	0.865	1.000	0.958	0.957	0.968	0.036
	S10-KS+	0.888	1.000	0.961	0.961	0.971	0.033
$(1+e^{-1})$-out	U-Max	0.251	1.000	0.952	0.945	0.967	0.081
	S1-Max	0.642	1.000	0.967	0.966	0.980	0.042
	S5-Max	0.918	1.000	0.977	0.977	0.985	0.020
	S10-Max	0.934	1.000	0.980	0.979	0.985	0.018
	S1-KS	0.642	1.000	0.963	0.962	0.972	0.041
	S5-KS	0.918	1.000	0.972	0.972	0.976	0.020
	S10-KS	0.934	1.000	0.975	0.975	0.977	0.018
	S1-KS+	0.857	1.000	0.972	0.972	0.979	0.025
	S5-KS+	0.925	1.000	0.978	0.978	0.984	0.018
	S10-KS+	0.939	1.000	0.980	0.980	0.985	0.016
2-out	U-Max	0.254	1.000	0.972	0.966	0.996	0.079
	S1-Max	0.652	1.000	0.987	0.986	0.999	0.036
	S5-Max	0.952	1.000	0.995	0.995	0.999	0.009
	S10-Max	0.968	1.000	0.996	0.996	1.000	0.007
	S1-KS	0.651	1.000	0.974	0.974	0.981	0.035
	S5-KS	0.945	1.000	0.982	0.982	0.984	0.013
	S10-KS	0.947	1.000	0.984	0.983	0.984	0.012
	S1-KS+	0.860	1.000	0.980	0.979	0.984	0.020
	S5-KS+	0.950	1.000	0.984	0.984	0.987	0.012
	S10-KS+	0.952	1.000	0.986	0.986	0.987	0.011

Table 4.5: For each matrix in the first dataset and each proposed heuristic, five runs are performed. The statistics are computed over the mean of these results; the minimum, maximum, arithmetic and geometric averages, median and standard deviation are reported.



(a) 1-out graphs (b) 2-out graphs

Figure 4.4: The matching qualities sorted in increasing order for Karp-Sipser on the original graph, S5-KS and S5-KS+ on 1-out and 2-out graphs. The figures also contain the quality of the maximum cardinality matchings in these graphs. The experiments are performed on the 115 graphs in the first dataset.

algorithm for these subgraphs.

As Table 4.5 shows, more scaling iterations increase the maximum matching cardinalities on k-out graphs. Although this is much more clear when the worst-case graphs are considered, it can also be observed for arithmetic and geometric means. Since U-Max is the no scaling case, the impact of the first scaling iteration (S1-KS vs U-Max) is significant. On the other hand, the difference on the matching quality for S5-KS and S10-KS is minor. Hence, five scaling iterations are deemed sufficient for the proposed heuristics in practice.

As the theory suggests, the heuristics St-KS perform well for $(1 + e^{-1})$-out and 2-out graphs. With $t \in \{5, 10\}$, their quality is almost on par with Karp-Sipser on the original graph, and even better for 2-out graphs. In addition, applying Karp-Sipser on the subgraph of unmatched vertices to obtain a maximal matching does not increase the matching quality much. Since this subgraph is small, the overhead of this extra work will not be significant. Furthermore, this extra step significantly improves the matching quality for 1-out graphs which a.a.s. do not have a perfect matching.

To better understand the practical performance of the proposed heuristics and the impact of the additional Karp-Sipser execution, we profile their performance by sorting their matching qualities in increasing order for all 115 matrices. Figure 4.4a plots these profiles on 1-out and 2-out graphs for the heuristics with five scaling iterations. As the first figure shows, five iterations are sufficient to obtain 0.86 matching quality except 1.7% of the 1-out experiments. The figure also shows that the maximum matching cardinality in a random 1-out graph is worse than what Karp-Sipser can obtain on the original graph. This is why although S5-KS finds maximum matchings on

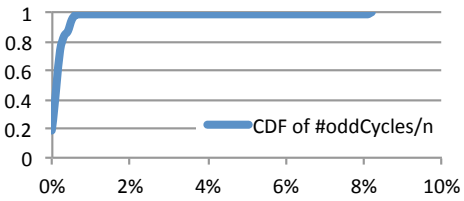


Figure 4.5: The cumulative distribution of the ratio of number of odd cycles remaining after the first phase of Karp-Sipser in One-Out to the number of vertices in the graph.

1-out graphs, its performance is still worse than Karp-Sipser. The additional Karp-Sipser in S5-KS+ closes almost all of this gap and makes the matching qualities close to those of Karp-Sipser. On the contrary, for 2-out graphs generated with five scaling iterations, the maximum matching cardinality is more than the cardinality of the matchings found by Karp-Sipser. There is still a gap between the best possible (red line) and what Karp-Sipser can find (blue line) on 2-out graphs. We believe that this gap can be targeted by specialized, efficient exact matching algorithms for 2-out graphs.

There are 30 rank-deficient matrices without total support among the 115 matrices in the first dataset. We observed that even for 1-out graphs, the worst-case quality for S5-KS is 0.86 and the average is 0.93. Hence, the proposed approach also works well for rank-deficient matrices/graphs in practice.

Since the number c of odd cycles at the end of the first phase of Karp-Sipser is a performance measure, we investigate it. For each matrix, we compute the ratio c/n. We then plot the cumulative distribution of these values in Fig. 4.5. For all the experiments except one, this ratio is less than 1%. For the extreme case, the ratio increases to 8%. In that particular case, the matching found in the 1-out subgraph is maximum for the input graph (i.e., the number of odd components is also large in the original graph).

Hard instances for Karp-Sipser

Let $\mathbf{A}$ be an $n \times n$ symmetric matrix, V_1 be the set of $\mathbf{A}$'s first $n/2$ vertices, and V_2 be the set of $\mathbf{A}$'s last $n/2$ vertices. As Figure 4.6 shows, $\mathbf{A}$ has a full $V_1 \times V_1$ block ignoring the diagonal, i.e., a clique, and an empty $V_2 \times V_2$ block. The last $h \ll n$ vertices of V_1 are connected to all the vertices in the corresponding graph. The block $V_1 \times V_2$ has a nonzero diagonal, hence the corresponding graph has a perfect matching. Note that the same instances are used for the bipartite case.

On such a matrix with $h = 0$, Karp-Sipser consumes the whole graph during Phase 1 and finds a maximum cardinality matching. When $h > 1$,

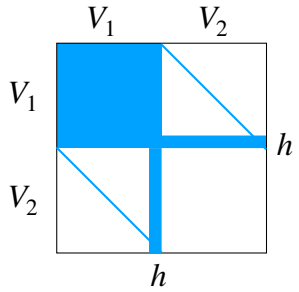


Figure 4.6: Adjacency matrices of hard undirected graph instances for Karp-Sipser.

		One-Out					
		5 iters.		10 iters.		20 iters.	
h	Karp-Sipser	error	quality	error	quality	error	quality
2	0.96	7.54	0.99	0.68	0.99	0.22	1.00
8	0.78	8.52	0.97	0.78	0.99	0.23	0.99
32	0.68	6.65	0.81	1.09	0.99	0.26	0.99
128	0.63	3.32	0.53	1.89	0.90	0.33	0.98
512	0.63	1.24	0.55	1.17	0.59	0.61	0.86

Table 4.6: Results for the hard instances with $n = 5000$ and different h values. One-Out is executed with 5, 10, and 20 scaling iterations and scaling errors are also reported. Averages of five are reported for each cell.

Phase 1 immediately ends, since there is no degree-one vertex. In Phase 2, the first edge (nonzero) consumed by Karp-Sipser is selected from a uniform distribution over the nonzeros whose corresponding rows and columns are still unmatched. Since the block $V_1 \times V_1$ forms a clique, it is more likely that the nonzero will be chosen from this block. Thus, a vertex in V_1 will be matched with another vertex from V_1, which is a bad decision since the block $V_2 \times V_2$ is completely empty. Hence, we expect a decrease on the performance of Karp-Sipser as h increases. On the other hand the probability that the proposed heuristics chooses an edge from that block goes to zero, as those entries cannot be in a perfect matching.

Table 4.6 shows that the quality of Karp-Sipser drops to 0.63 as h increases. In comparison, One-Out heuristic with five scaling iterations maintains a good quality for small h values. However, a better scaling with more iterations (10 and 20 for $h = 128$ and $h = 512$, respectively) is required to guarantee the desired matching quality—see the scaling error in the table.

Experiments on large-scale graphs

These experiments are performed on a machine running 64-bit CentOS 6.5, which has 30 cores each of which is an Intel Xeon CPU E7-4870 v2 core

Matrix	$\|V\|$	$\|E\|$	Karp-Sipser *Quality*	Karp-Sipser *time*
`cage15`	5.2	94.0	1.00	5.82
`dielFilterV3real`	1.1	88.2	0.99	3.36
`hollywood-2009`	1.1	112.8	0.93	4.18
`nlpkkt240`	28.0	746.5	0.98	52.95
`rgg_n_2_24_s0`	16.8	265.1	0.98	19.49

	One-Out-S5-KS+						Two-Out-S5-KS+					
		Execution time (seconds)						Execution time (seconds)				
Matrix	*Quality*	*Scale*	*1-out*	*KS*	*KS+*	Total	*Quality*	*Scale*	*2-out*	*KS*	*KS+*	Total
`cage`	0.93	0.67	0.85	0.65	0.05	2.21	0.99	0.67	1.48	1.78	0.01	3.94
`diel`	0.98	0.52	0.36	0.07	0.01	0.95	0.99	0.52	0.62	0.16	0.00	1.30
`holly`	0.91	1.12	0.45	0.06	0.02	1.65	0.95	1.12	0.76	0.13	0.01	2.01
`nlpk`	0.93	4.44	4.99	3.96	0.27	13.66	0.98	4.44	9.43	10.56	0.11	24.54
`rgg`	0.93	2.33	2.33	2.08	0.17	6.91	0.98	2.33	4.01	6.70	0.11	13.15

Table 4.7: Summary of the results with five large-scale matrices for original Karp-Sipser and the proposed One-Out and TwoOut heuristics with five scaling iterations and post-processing for maximal matchings. Matrix names are shortened in the lower half.

operating at 2.30 GHz. To choose five large-scale matrices from SuiteSparse, we first sorted the pattern-symmetric matrices in the collection in decreasing order of their nonzero count. We then chose the five largest matrices from different families to increase the variety of experiments. The details of these matrices are given in Table 4.7. This table also contains the run times and matching qualities of the original Karp-Sipser, and the proposed One-Out and Two-Out heuristics. The proposed heuristics have five scaling iterations and also apply Karp-Sipser at the end for ensuring maximality.

The run time of the proposed heuristics are analyzed in four stages in Table 4.7. The *Scale* stage scales the matrix with five iterations, the *k-out* stage chooses the neighbors and constructs the k-out subgraph, the *KS* stage applies Karp-Sipser on the k-out subgraph, and *KS+* is the stage for the additional Karp-Sipser at the end. The total time with these four stages is also shown for each instance. The quality results are consistent with the results obtained on the first dataset. As the table shows, the proposed heuristics are faster than the original Karp-Sipser on the input graph. For 1-out graphs, the proposed approach is 2.5–3.9 faster than Karp-Sipser on the original graph. The speedups are in between 1.5–2.6 for 2-out graphs with five iterations.

4.4 Summary, further notes and references

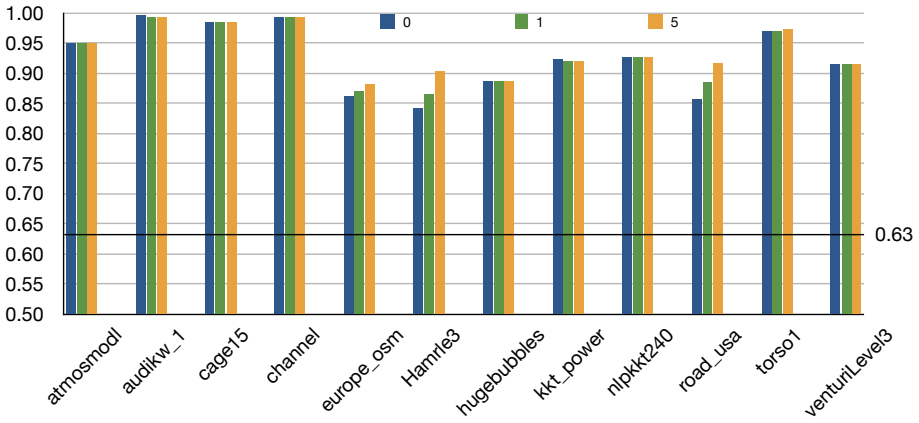
We investigated two heuristics for the bipartite maximum cardinality matching problem. The first one, OneSidedMatch, is shown to have an approx-

imation guarantee of $1 - 1/e \approx 0.632$. The second heuristic, TwoSidedMatch, is shown to have an approximation guarantee of 0.866. Both algorithms use well-known methods to scale the sparse matrix associated with the given bipartite graph to a doubly stochastic form whose entries are used as the probability density functions to randomly select a subset of the edges of the input graph. OneSidedMatch selects exactly n edges to construct a subgraph in which a matching of the guaranteed cardinality is identified with virtually no overhead, both in sequential and parallel execution. TwoSidedMatch selects around $2n$ edges and then runs the Karp-Sipser heuristic as an exact algorithm on the selected subgraph to obtain a matching of conjectured cardinality. The subgraphs are analyzed to develop a specialized Karp-Sipser algorithm for efficient parallelization. All theoretical investigations are first performed assuming bipartite graphs with perfect matchings, and the scaling algorithms have converged. Then, theoretical arguments and experimental evidence are provided to extend the results to cover other cases and validate the applicability and practicality of the proposed heuristics in general settings.

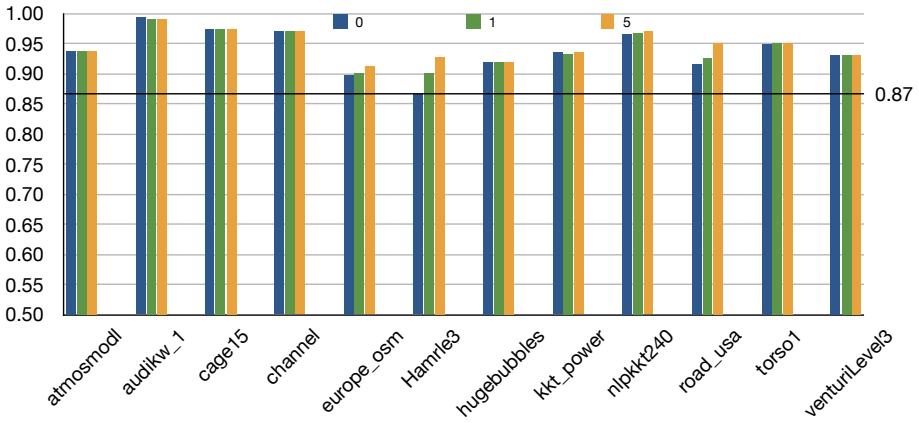
The proposed OneSidedMatch and TwoSidedMatch heuristics do not guarantee maximality of the matching found. One can visit the edges of the unmatched row vertices and match them to an available neighbor. We have carried out this greedy post-process and obtained the results shown in Figures 4.7a and 4.7b, in comparison with what was shown in Figures 4.2a and 4.2b. We see that the greedy post-process makes significant difference for OneSidedMatch; the approximation ratios achieved are well beyond the bound 0.632 and are close to those of TwoSidedMatch. This observation attests to the efficiency of the whole approach.

We also extended the heuristics and their analysis for approximating the maximum cardinality matchings on general (undirected) graphs. Since we have in the general case one class of vertices, the 1-out based approach corresponds to the TwoSidedMatch heuristics in the bipartite case. Theoretical analysis, which can be perceived as an analysis of Karp-Sipser on the random 1-out graph model, showed that the approximation guarantee is slightly less than 0.866. The losses with respect to the bipartite case are due to the existence of odd-length cycles. Our experiments suggest that one of the heuristics obtains as good results as Karp-Sipser while being faster and more reliable.

A more rigorous treatment and elaboration of the variants of the heuristics for general graphs seem worthwhile. Although Karp-Sipser works well for these graphs, we wonder if there are exact, linear time algorithms for $(1 + e^{-1})$-out and 2-out graphs. We also plan to work on the parallel implementations of these algorithms.



(a) OneSidedMatch



(b) TwoSidedMatch

Figure 4.7: Matching qualities obtained after making the matchings found by OneSidedMatch and TwoSidedMatch maximal. The horizontal lines are at 0.632 and 0.866, respectively, which are the approximation guarantees for the heuristics. The legends contain the number of scaling iterations. Compare with Figures 4.2a and 4.2b where maximality of the matchings was not guaranteed.

Chapter 5

Birkhoff-von Neumann decomposition of doubly stochastic matrices

In this chapter, we investigate the Birkhoff-von Neumann (BvN) decomposition described in Section 1.4. The main problem that we address is restated below for convenience.

MinBvNDec: **A BvN decomposition with the minimum number of permutation matrices.** Given a doubly stochastic matrix $\mathbf{A}$, find a BvN decomposition

$$\mathbf{A} = \alpha_1 \mathbf{P}_1 + \alpha_2 \mathbf{P}_2 + \cdots + \alpha_k \mathbf{P}_k \,. \tag{5.1}$$

of $\mathbf{A}$ with the minimum number k of permutation matrices.

We first show that the MinBvNDec problem is NP-hard. We then propose a heuristic for obtaining a BvN decomposition with a small number of permutation matrices. We investigate some of the properties of the heuristic theoretically and experimentally. In this chapter, we also give a generalization of the BvN decomposition for real matrices with possibly negative entries and use this generalization for designing preconditioners for iterative linear system solvers.

5.1 The minimum number of permutation matrices

We show in this section that the decision version of the problem is NP-complete. We first give some definitions and preliminary results.

5.1.1 Preliminaries

A multi-set can contain duplicate members. Two multi-sets are equivalent if they have the same set of members with the same number of repetitions.

Let $\mathbf{A}$ and $\mathbf{B}$ be two $n \times n$ matrices. We write $\mathbf{A} \subseteq \mathbf{B}$ to denote that for each nonzero entry of $\mathbf{A}$, the corresponding entry of $\mathbf{B}$ is nonzero. In particular, if $\mathbf{P}$ is an $n \times n$ permutation matrix and $\mathbf{A}$ is a nonnegative $n \times n$ matrix, $\mathbf{P} \subseteq \mathbf{A}$ also means that the entries of $\mathbf{A}$ at the positions corresponding to the nonzeros of $\mathbf{P}$ are positive. We use $\mathbf{P} \odot \mathbf{A}$ to denote the entrywise product of $\mathbf{P}$ and $\mathbf{A}$, which selects the entries of $\mathbf{A}$ at the positions corresponding to the nonzero entries of $\mathbf{P}$. We also use $\min\{\mathbf{P} \odot \mathbf{A}\}$ to denote the minimum entry of $\mathbf{A}$ at the nonzero positions of $\mathbf{P}$.

Let U be a set of positions of the nonzeros of $\mathbf{A}$. Then U is called strongly stable [31], if for each permutation matrix $\mathbf{P} \subseteq \mathbf{A}$, $p_{kl} = 1$ for at most one pair $(k, l) \in U$.

Lemma 5.1 (Brualdi [32])**.** *Let $\mathbf{A}$ be a doubly stochastic matrix. Then, in a BvN decomposition of $\mathbf{A}$, there are at least $\gamma(\mathbf{A})$ permutation matrices, where $\gamma(\mathbf{A})$ is the maximum cardinality of a strongly stable set of positions of $\mathbf{A}$.*

Note that $\gamma(\mathbf{A})$ is no smaller than the maximum number of nonzeros in a row or a column of $\mathbf{A}$ for any matrix $\mathbf{A}$. Brualdi [32] shows that for any integer t with $1 \leq t \leq \lceil n/2 \rceil \lceil (n+1)/2 \rceil$, there exists an $n \times n$ doubly stochastic matrix $\mathbf{A}$ such that $\gamma(\mathbf{A}) = t$.

An $n \times n$ circulant matrix $\mathbf{C}$ is defined as follows. The first row of $\mathbf{C}$ is specified as $c_1, \ldots, c_n$, and the ith row is obtained from the $(i-1)$th one by a cyclic rotation to the right, for $i = 2, \ldots, n$:

$$\mathbf{C} = \begin{bmatrix} c_1 & c_2 & \cdots & c_n \\ c_n & c_1 & \cdots & c_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_2 & c_3 & \cdots & c_1 \end{bmatrix}.$$

We state and prove an obvious lemma to be used later.

Lemma 5.2. *Let $\mathbf{C}$ be an $n \times n$ positive circulant matrix whose first row is $c_1, \ldots, c_n$. The matrix $\mathbf{C}' = \frac{1}{\sum c_j}\mathbf{C}$ is doubly stochastic, and all BvN decompositions with n permutation matrices have the same multi-set of coefficients $\{\frac{c_i}{\sum c_j} : \text{for } i = 1, \ldots, n\}$.*

Proof. Since the first row of $\mathbf{C}'$ has all nonzero entries and only n permutation matrices are permitted, the multi-set of coefficients must be the entries in the first row. □

In the lemma, if $c_i = c_j$ for some $i \neq j$, we will have the same value c_i/c for two different permutation matrices. As a sample BvN decomposition, let $c = \sum c_i$ and consider $\frac{1}{c}\mathbf{C} = \sum \frac{c_j}{c}\mathbf{D}_j$, where $\mathbf{D}_j$ is the permutation matrix corresponding to the $(j-1)$th diagonal: for $j = 1, \ldots, n$, the matrix $\mathbf{D}_j$ has 1s at the positions $(i, i+j-1)$ for $i = 1, \ldots, n-j+1$ and at the positions $(n-j+1+k, k)$ for $k = 1, \ldots, j-1$, where we assumed that the second set is void for $j = 1$.

Note that the lemma is not concerned by the uniqueness of the BvN decomposition which would need a unique set of permutation matrices as well. If all c_is were distinct, this would have been true, where the set of $\mathbf{D}_j$s described above would define the unique permutation matrices. Also, a more general variant of the lemma concerns the decompositions whose cardinality k is equal to the maximum cardinality of a strongly stable set. In this case too, the coefficients in the decomposition will correspond to the entries in a strongly stable set of cardinality k.

5.1.2 The computational complexity of MinBvNDec

Here, we prove that the MinBvNDec problem is NP-complete in the strong sense, i.e., it remains NP-complete when the instance is represented in unary. It suffices to do a reduction from a strongly NP-complete problem to prove the strong NP-completeness [22, Section 6.6].

Theorem 5.1. *The problem of deciding whether there is a Birkhoff-von Neumann decomposition of a given doubly stochastic matrix with k permutation matrices is strongly NP-complete.*

Proof. It is clear that the problem belongs to NP, as it is easy to check in polynomial time that a given decomposition is equal to a given matrix.

To establish NP-completeness, we demonstrate a reduction from the well-known 3-PARTITION problem which is NP-complete in the strong sense [93, p. 96]. Consider an instance of 3-PARTITION: given an array A of $3m$ positive integers, a positive integer B such that $\sum_{i=1}^{3m} a_i = mB$, and for all i it holds that $B/4 < a_i < B/2$, does there exist a partition of A into m disjoint arrays $S_1, \ldots, S_m$ such that each S_i has three elements whose sum is B. Let $\mathcal{I}_1$ denote an instance of 3-PARTITION.

We build the following instance $\mathcal{I}_2$ of MinBvNDec corresponding to $\mathcal{I}_1$ given above. Let

$$\mathbf{M} = \begin{bmatrix} \frac{1}{m}\mathbf{E}_m & O \\ O & \frac{1}{mB}\mathbf{C} \end{bmatrix}$$

where $\mathbf{E}_m$ is an $m \times m$ matrix whose entries are 1, and $\mathbf{C}$ is an $3m \times 3m$ circulant matrix whose first row is $a_1, \ldots, a_{3m}$. It is easy to see that $\mathbf{M}$ is doubly stochastic. A solution of $\mathcal{I}_2$ is a BvN decomposition of $\mathbf{M}$ with $k = 3m$ permutations. We will show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ has a solution.

Assume that $\mathcal{I}_1$ has a solution with $S_1, \ldots, S_m$. Let $S_i = \{a_{i,1}, a_{i,2}, a_{i,3}\}$ and observe that $\frac{1}{mB}(a_{i,1} + a_{i,2} + a_{i,3}) = 1/m$. We identify three permutation matrices $\mathbf{P}_{i,d}$ for $d = 1, 2, 3$ in $\mathbf{C}$ which contain $a_{i,1}$, $a_{i,2}$ and $a_{i,3}$, respectively. We can write $\frac{1}{m}\mathbf{E}_m = \sum_{i=1}^{m} \frac{1}{m}\mathbf{D}_i$ where $\mathbf{D}_i$ is the permutation matrix corresponding to the $(i-1)$th diagonal (described after the proof of Lemma 5.2). We prepend $\mathbf{D}_i$ to the three permutation matrices $\mathbf{P}_{i,d}$ for $d = 1, 2, 3$ and obtain three permutation matrices for $\mathbf{M}$. We associate these three permutation matrices with $\alpha_{i,d} = \frac{a_{i,d}}{mB}$. Therefore, we can write

$$\mathbf{M} = \sum_{i=1}^{m} \sum_{d=1}^{3} \alpha_{i,d} \begin{bmatrix} \mathbf{D}_i & \\ & \mathbf{P}_{i,d} \end{bmatrix},$$

and obtain a BvN decomposition of $\mathbf{M}$ with $3m$ permutation matrices.

Assume that $\mathcal{I}_2$ has a BvN decomposition with $3m$ permutation matrices. This also defines two BvN decompositions with $3m$ permutation matrices for $\frac{1}{m}\mathbf{E}_m$ and $\frac{1}{mB}\mathbf{C}$. We now establish a correspondence between these two BvN's to finish the proof. Since $\frac{1}{mB}\mathbf{C}$ is a circulant matrix with $3m$ nonzeros in a row, any BvN decomposition of it with $3m$ permutation matrices has the coefficients $\frac{a_i}{mB}$ for $i = 1, \ldots, 3m$ by Lemma 5.2. Since $a_i + a_j < B$, we have $\frac{a_i + a_j}{mB} < 1/m$. Therefore, each entry in $\frac{1}{m}\mathbf{E}_m$ needs to be included in at least three permutation matrices. A total of $3m$ permutation matrices covers any row of $\frac{1}{m}\mathbf{E}_m$, say the first one. Therefore, for each entry in this row we have $\frac{1}{m} = \alpha_i + \alpha_j + \alpha_k$, for $i \neq j \neq k$ corresponding to three coefficients used in the BvN decomposition of $\frac{1}{mB}\mathbf{C}$. Note that these three indices i, j, k defining the three coefficients used for one entry of $\mathbf{E}_m$ cannot be used for another entry in the same row. This correspondence defines a partition of the $3m$ numbers α_i for $i = 1, \ldots, 3m$ into m groups with three elements each, where each group has a sum of $1/m$. The corresponding three a_i's in a group therefore sums up to B, and we have a solution to $\mathcal{I}_1$, concluding the proof. □

5.2 A result on the polytope of BvN decompositions

Let $\boldsymbol{\mathcal{S}}(\mathbf{A})$ be the polytope of all BvN decompositions for a given, doubly stochastic matrix $\mathbf{A}$. The extreme points of $\boldsymbol{\mathcal{S}}(\mathbf{A})$ are the ones that cannot be represented as a convex combination of other decompositions. Brualdi [32, pp. 197–198] observes that any generalized Birkhoff heuristic obtains an extreme point of $\boldsymbol{\mathcal{S}}(\mathbf{A})$, and predicts that there are extreme points of $\boldsymbol{\mathcal{S}}(\mathbf{A})$ which cannot be obtained by a generalized Birkhoff heuristic. In this section, we substantiate this claim by showing an example.

Any BvN decomposition of a given matrix $\mathbf{A}$ with the smallest number of permutation matrices is an extreme point of $\boldsymbol{\mathcal{S}}(\mathbf{A})$; otherwise the other

BvN decompositions expressing the said point would have smaller number of permutation matrices.

Lemma 5.3. *There are doubly stochastic matrices whose polytopes of BvN decompositions contain extreme points that cannot be obtained by a generalized Birkhoff heuristic.*

We will prove the lemma by giving an example. We use computational tools based on a mixed integer linear programming (MILP) formulation of the problem of finding a BvN decomposition with the smallest number of permutation matrices. We first describe the MILP formulation.

Let $\mathbf{A}$ be a given $n \times n$ doubly stochastic matrix, and Ω_n be the set of all $n \times n$ permutation matrices. There are $n!$ matrices in Ω_n; for brevity, let us refer to these permutations by $\mathbf{P}_1, \ldots, \mathbf{P}_{n!}$. We associate an incidence matrix $\mathbf{M}$ of size $n^2 \times n!$ with Ω_n. We fix an ordering of the entries of $\mathbf{A}$ so that each row of $\mathbf{M}$ corresponds to a unique entry in $\mathbf{A}$. Each column of $\mathbf{M}$ corresponds to a unique permutation matrix in Ω_n. We set $m_{ij} = 1$ if the ith entry of $\mathbf{A}$ appears in the permutation matrix $\mathbf{P}_j$, and set $m_{ij} = 0$ otherwise. Let $\vec{a}$ be the n^2-vector containing the values of the entries of $\mathbf{A}$ in the fixed order. Let $\vec{x}$ be a vector of $n!$ elements, x_j corresponding to the permutation matrix $\mathbf{P}_j$. With these definitions, MILP formulation for finding a BvN decomposition of $\mathbf{A}$ with the smallest number of permutation matrices can be written as:

$$\text{minimize} \sum_{j=1}^{n!} s_j \tag{5.2}$$

$$\text{subject to } \mathbf{M}\vec{x} = \vec{a}\,, \tag{5.3}$$

$$1 \geq x_j \geq 0, \quad \text{for } j = 1, \ldots, n!\,, \tag{5.4}$$

$$\sum_{j=1}^{n!} x_j = 1, \tag{5.5}$$

$$s_j \geq x_j, \qquad \text{for } j = 1, \ldots, n!\,, \tag{5.6}$$

$$s_j \in \{0, 1\}, \quad \text{for } j = 1, \ldots, n!\,. \tag{5.7}$$

In this MILP, s_j is a binary variable which is 1 only if $x_j > 0$, otherwise 0. The equality (5.3), the inequalities (5.4), and the equality (5.5) guarantee that we have a BvN decomposition of $\mathbf{A}$. This MILP can be used only for small problems. In this MILP, we can exclude any permutation matrix $\mathbf{P}_j$ from a decomposition by setting $s_j = 0$.

Proof of Lemma 5.3. Let the following 10 letters correspond to the numbers underneath

a	b	c	d	e	f	g	h	i	j
1	2	4	8	16	32	64	128	256	512

.

Consider the following matrix whose row sums and column sums are 1023, hence can be considered as doubly stochastic

$$\mathbf{A} = \begin{pmatrix} a+b & d+i & c+h & e+j & f+g \\ e+g & a+c & b+i & d+f & h+j \\ f+j & e+h & d+g & b+c & a+i \\ d+h & b+f & a+j & g+i & c+e \\ c+i & g+j & e+f & a+h & b+d \end{pmatrix}. \tag{5.8}$$

Observe that the entries containing the term 2^ℓ form a permutation matrix, for $\ell = 0, \ldots, 9$. Therefore, this matrix has a BvN decomposition with 10 permutation matrices. We created the MILP above and found 10 as the smallest number of permutation matrices by calling the CPLEX solver [1] via the NEOS Server [58, 68, 100]. Hence the described decomposition is an extreme point of $\boldsymbol{\mathcal{S}}(\mathbf{A})$. None of the said permutation matrices annihilate any entry of the matrix. Therefore, at the first step no entry of $\mathbf{A}$ gets reduced to zero, regardless of the order of permutation matrices. Thus, this decomposition cannot be obtained by a generalized Birkhoff heuristic.

One can create a family of matrices with arbitrary sizes by embedding the same matrix into a larger one of the form

$$\mathbf{B} = \begin{pmatrix} 1023 \cdot \mathbf{I} & O \\ O & \mathbf{A} \end{pmatrix},$$

where $\mathbf{I}$ is the identity matrix with the desired size. All BvN decompositions of $\mathbf{B}$ can be obtained by extending the permutation matrices in $\mathbf{A}$'s BvN decompositions with $\mathbf{I}$. That is, for a permutation matrix $\mathbf{P}$ in a BvN decomposition of $\mathbf{A}$, $\begin{pmatrix} \mathbf{I} & O \\ O & \mathbf{P} \end{pmatrix}$ is a permutation matrix in the corresponding BvN decomposition of $\mathbf{B}$. Furthermore, all permutation matrices in an arbitrary BvN decomposition of $\mathbf{B}$ must have $\mathbf{I}$ as the principle sub-matrix, and the rest should correspond to a permutation matrix in $\mathbf{A}$, defining a BvN decomposition for $\mathbf{A}$. Hence, the extreme point $\boldsymbol{\mathcal{S}}(\mathbf{B})$ corresponding to the extreme point of $\boldsymbol{\mathcal{S}}(\mathbf{A})$ with 10 permutation matrices cannot be found by a generalized Birkhoff heuristic. □

Let us investigate the matrix $\mathbf{A}$ and its BvN decomposition given in the proof of Lemma 5.3. Let $\mathbf{P}_a, \mathbf{P}_b, \ldots, \mathbf{P}_j$ be the 10 permutation matrices of the decomposition, corresponding to $a, b, \ldots, j$. We solve 10 MILPs in which we set $s_t = 0$ for one $t \in \{a, b, \ldots, j\}$. This way, we try to find a BvN decomposition of $\mathbf{A}$ without $\mathbf{P}_a$, without $\mathbf{P}_b$ and so on, always with the smallest number of permutation matrices. The smallest number of permutation matrices in these 10 MILPs were 11. This certifies that the only BvN decomposition with 10 permutation matrices necessarily contains $\mathbf{P}_a, \mathbf{P}_b, \ldots, \mathbf{P}_j$. It is easy to see that there is a unique solution to the equality $\mathbf{M}\vec{x} = \vec{a}$ of

$$\mathbf{A} = \begin{pmatrix} 3 & 264 & 132 & 528 & 96 \\ 80 & 5 & 258 & 40 & 640 \\ 544 & 144 & 72 & 6 & 257 \\ 136 & 34 & 513 & 320 & 20 \\ 260 & 576 & 48 & 129 & 10 \end{pmatrix}$$

(a) The sample matrix

129	511	257	63	33	15	7	3	2	2	1
3	4	2	5	**5**	4	**2**	**3**	**4**	1	**1**
5	**5**	3	1	4	1	**4**	2	**1**	**2**	**3**
2	1	**5**	3	**1**	**2**	3	4	**3**	4	**4**
1	3	4	**4**	2	5	**1**	5	**5**	**3**	**2**
4	2	1	2	3	**3**	5	**1**	**2**	5	**5**

(b) A BvN decomposition

Figure 5.1: The matrix $\mathbf{A}$ from Lemma 5.3, and a BvN decomposition with 11 permutation matrices which can be obtained by a generalized Birkhoff heuristic. Each column in (b) corresponds to a permutation, where the first line gives the associated coefficient, and the following lines give the column indices matched to the rows 1 to 5 of $\mathbf{A}$.

the MILP with $x_t = 0$ for $t \notin \{a, b, \ldots, j\}$, as the submatrix $\mathbf{M}$ containing only the corresponding 10 columns has full column rank.

Any generalized Birkhoff heuristic obtains at least 11 permutation matrices for the matrix $\mathbf{A}$ of the proof of Lemma 5.3. One such decomposition is shown in a tabular format in Fig. 5.1. In Fig. 5.1a, we write the matrix of the lemma explicitly for convenience. Then in Fig. 5.1b, we give a BvN decomposition. The column headers (the first line) in the table contain the coefficients of the permutation matrices. The nonzero column indices of the permutation matrices are stored by rows. For example, the first permutation has the coefficient 129 and columns 3, 5, 2, 1, and 4 are matched to the rows 1–5. The bold indices signify entries whose values are equivalent to the coefficients of the corresponding permutation matrices, at the time where the permutation matrices are found. Therefore, the corresponding entries become zero after the corresponding step. For example, in the first permutation, $a_{5,4} = 129$.

The output of $\text{Greedy}_{\text{BvN}}$ for the matrix $\mathbf{A}$ is given in Fig. 5.2 for reference. It contains 12 permutation matrices.

5.3 Two heuristics

There are heuristics to compute a BvN decomposition for a given matrix $\mathbf{A}$. In particular, the following family of heuristics is based on the constructive proof of Birkhoff. Let $\mathbf{A}^{(0)} = \mathbf{A}$. At every step $j \geq 1$, find a permutation matrix $\mathbf{P}_j$ having its ones at the positions of the nonzero elements of $\mathbf{A}^{(j-1)}$, use the minimum nonzero element of $\mathbf{A}^{(j-1)}$ at the positions identified by $\mathbf{P}_j$ as α_j, set $\mathbf{A}^{(j)} = \mathbf{A}^{(j-1)} - \alpha_j \mathbf{P}_j$, and repeat the computations in the next step $j + 1$ until $\mathbf{A}^{(j)}$ becomes void. Any heuristic of this type is called *generalized Birkhoff heuristic*.

$$\mathbf{A} = 513 \cdot \begin{bmatrix} 0&0&0&1&0\\0&0&0&0&1\\1&0&0&0&0\\0&0&1&0&0\\0&1&0&0&0 \end{bmatrix} + 257 \cdot \begin{bmatrix} 0&1&0&0&0\\0&0&1&0&0\\0&0&0&0&1\\0&0&0&1&0\\1&0&0&0&0 \end{bmatrix} + 127 \cdot \begin{bmatrix} 0&0&1&0&0\\0&0&0&0&1\\0&1&0&0&0\\1&0&0&0&0\\0&0&0&1&0 \end{bmatrix} + 63 \cdot \begin{bmatrix} 0&0&0&0&1\\1&0&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&1&0&0&0 \end{bmatrix} +$$
$$31 \cdot \begin{bmatrix} 0&0&0&0&1\\0&0&0&1&0\\1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0 \end{bmatrix} + 15 \cdot \begin{bmatrix} 0&0&0&1&0\\1&0&0&0&0\\0&1&0&0&0\\0&0&0&0&1\\0&0&1&0&0 \end{bmatrix} + 7 \cdot \begin{bmatrix} 0&1&0&0&0\\0&0&0&1&0\\0&0&1&0&0\\1&0&0&0&0\\0&0&0&0&1 \end{bmatrix} + 3 \cdot \begin{bmatrix} 0&0&1&0&0\\0&1&0&0&0\\0&0&0&1&0\\0&0&0&0&1\\1&0&0&0&0 \end{bmatrix} +$$
$$2 \cdot \begin{bmatrix} 0&0&0&0&1\\0&0&0&1&0\\0&1&0&0&0\\1&0&0&0&0\\0&0&1&0&0 \end{bmatrix} + 2 \cdot \begin{bmatrix} 0&0&1&0&0\\1&0&0&0&0\\0&0&0&1&0\\0&1&0&0&0\\0&0&0&0&1 \end{bmatrix} + 2 \cdot \begin{bmatrix} 1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&0&1\\0&0&0&1&0 \end{bmatrix} + 1 \cdot \begin{bmatrix} 1&0&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&1&0&0&0\\0&0&0&0&1 \end{bmatrix}$$

Figure 5.2: The output of $\text{Greedy}_{\text{BvN}}$ for the matrix given in the proof of Lemma 5.3.

Given an $n \times n$ matrix $\mathbf{A}$, we can associate a bipartite graph $G_{\mathbf{A}} = (R \cup C, E)$ to it. The rows of $\mathbf{A}$ correspond to the vertices in the set R, and the columns of $\mathbf{A}$ correspond to the vertices in the set C so that $(r_i, c_j) \in E$ iff $a_{ij} \neq 0$. A perfect matching in $G_{\mathbf{A}}$, or G in short, is a set of n edges no two sharing a row or a column vertex. Therefore a perfect matching $\mathcal{M}$ in G defines a unique permutation matrix $\mathbf{P}_{\mathcal{M}} \subseteq \mathbf{A}$.

Birkhoff's heuristic: This is the original heuristic used in proving that a doubly stochastic matrix has a BvN decomposition, described for example by Brualdi [32]. Let a be the smallest nonzero of $\mathbf{A}^{(j-1)}$ and $G^{(j-1)}$ be the bipartite graph associated with $\mathbf{A}^{(j-1)}$. Find a perfect matching $\mathcal{M}$ in $G^{(j-1)}$ containing a, set $\alpha_j \leftarrow a$ and $\mathbf{P}_j \leftarrow \mathbf{P}_{\mathcal{M}}$.

$\text{Greedy}_{\text{BvN}}$ **heuristic:** At every step j, among all perfect matchings in $G^{(j-1)}$ find one whose minimum element is the maximum. That is, find a perfect matching $\mathcal{M}$ in $G^{(j-1)}$ where $\min\{\mathbf{P}_{\mathcal{M}} \odot \mathbf{A}^{(j-1)}\}$ is the maximum. This "bottleneck" matching problem is polynomial time solvable, with for example MC64 [72]. In this greedy approach, α_j is the largest amount we can subtract from a row and a column of $\mathbf{A}^{(j-1)}$, and we hope to obtain a small k.

Algorithm 11: $\text{Greedy}_{\text{BvN}}$: A greedy heuristic for constructing a BvN decomposition

Data: $\mathbf{A}$: a doubly stochastic matrix
Result: a BvN decomposition

```
k ← 0
while nnz(A) > 0 do
    k ← k + 1
    P_k the pattern of a bottleneck perfect matching M in A
    α_k ← min{P_k ⊙ A^(j−1)}
    A ← A − α_k P_k
```

$$\mathbf{A}^{(0)} = \begin{pmatrix} \mathbf{1} & 4 & 1 & 0 & 0 & 0 \\ 0 & 1 & \mathbf{4} & 1 & 0 & 0 \\ 0 & 0 & 1 & \mathbf{4} & 1 & 0 \\ 0 & 0 & 0 & 1 & \mathbf{4} & 1 \\ 1 & 0 & 0 & 0 & 1 & \mathbf{4} \\ 4 & \mathbf{1} & 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{A}^{(1)} = \begin{pmatrix} 0 & 4 & \mathbf{1} & 0 & 0 & 0 \\ 0 & \mathbf{1} & 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & \mathbf{3} & 1 & 0 \\ 0 & 0 & 0 & 1 & \mathbf{3} & 1 \\ 1 & 0 & 0 & 0 & 1 & \mathbf{3} \\ \mathbf{4} & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{A}^{(2)} = \begin{pmatrix} 0 & \mathbf{4} & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & \mathbf{1} & 0 & 0 \\ 0 & 0 & \mathbf{1} & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 & \mathbf{2} & 1 \\ 1 & 0 & 0 & 0 & 1 & \mathbf{2} \\ \mathbf{3} & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{A}^{(3)} = \begin{pmatrix} 0 & \mathbf{3} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{3} & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & \mathbf{1} & 0 \\ 0 & 0 & 0 & \mathbf{1} & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & \mathbf{1} \\ \mathbf{2} & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{A}^{(4)} = \begin{pmatrix} 0 & \mathbf{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \mathbf{1} \\ 1 & 0 & 0 & 0 & \mathbf{1} & 0 \\ \mathbf{1} & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{A}^{(5)} = \begin{pmatrix} 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} \end{pmatrix}$$

Figure 5.3: A sample matrix to show that the original Birkhoff heuristic can obtain BvN decomposition with n permutation matrices while the optimum one has 3.

Analysis of the two heuristics

As we will see in Section 5.4, Greedy$_{\text{BvN}}$ can obtain a much smaller number of permutation matrices compared to the Birkhoff heuristic. Here, we compare these two heuristics theoretically. First we show that the original Birkhoff heuristic does not have any constant ratio approximation guarantee. Furthermore, for an $n \times n$ matrix, its worst-case approximation ratio is $\Omega(n)$.

We begin with a small example shown in Fig. 5.3. We decompose a 6×6 matrix $\mathbf{A}^{(0)}$ which has an optimal BvN decomposition with three permutation matrices; the main diagonal, the one containing the entries equal to 4, and the one containing the remaining entries. For simplicity, we used integer values in our example. However, since the row and column sums of $\mathbf{A}^{(0)}$ is equal to 6, it can be converted to a doubly stochastic matrix by dividing all the entries to 6. Instead of the optimal decomposition, in the figure, we obtain the permutation matrices as the original Birkhoff heuristic does. Each red-colored entry set is a permutation and contains the minimum possible value, 1.

In the following, we show how to generalize the idea for having matrices of arbitrarily large size, with three permutation matrices in an optimal decomposition, while the original Birkhoff heuristic obtains n permutation

matrices.

Lemma 5.4. *The worst-case approximation ratio of the Birkhoff heuristic is* $\Omega(n)$.

Proof. For any given integer $n \geq 3$, we show that there is a matrix of size $n \times n$ whose optimal BvN decomposition has 3 permutations, whereas the Birkhoff heuristic obtains a BvN decomposition with exactly n permutation matrices. The example in Fig. 5.3 is a special case for $n = 6$ for the following construction process.

Let $f : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ be the function $f(x) = (x \bmod n) + 1$. Given a matrix $\mathbf{M}$, let $\mathbf{M}' = \mathcal{F}(\mathbf{M})$ be another matrix containing the same set of entries where the function $f(\cdot)$ is used on the coordinate indices to redistribute the entries of $\mathbf{M}$ on $\mathbf{M}'$. That is $m_{i,j} = m'_{f(i),f(j)}$. Since $f(\cdot)$ is one-to-one and onto, if $\mathbf{M}$ is a permutation matrix then $\mathcal{F}(\mathbf{M})$ is also a permutation matrix. We will start with a permutation matrix, and run it through $\mathcal{F}$ for $n - 1$ times to obtain n permutation matrices, which are all different. By adding these permutation matrices, we will obtain a matrix $\mathbf{A}$ whose optimal BvN decomposition has three permutation matrices, while the n permutation matrices used to create $\mathbf{A}$ correspond to a decomposition that can be obtained by the Birkhoff heuristic.

Let $\mathbf{P}_1$ be the permutation matrix whose ones, which are partitioned into three sets, are at the positions

$$\overbrace{(1,1)}^{\text{1st set}}, \overbrace{(n,2)}^{\text{2nd set}}, \overbrace{(2,3),(3,4),\ldots,(n-1,n)}^{\text{3rd set}} . \tag{5.9}$$

Let us use $\mathcal{F}(\cdot)$ to generate a matrix sequence $\mathbf{P}_i = \mathcal{F}(\mathbf{P}_{i-1})$ for $2 \leq i \leq n$. For example, $\mathbf{P}_2$'s nonzeros are at the positions

$$\overbrace{(2,2)}^{\text{1st set}}, \overbrace{(1,3)}^{\text{2nd set}}, \overbrace{(3,4),(4,5),\ldots,(n,1)}^{\text{3rd set}} .$$

We then add the $\mathbf{P}_i$s to build the matrix

$$\mathbf{A} = \mathbf{P}_1 + \mathbf{P}_2 + \cdots + \mathbf{P}_n .$$

We have the following observations about the nonzero elements of $\mathbf{A}$:

1. $a_{i,i} = 1$ for all $i = 1, \ldots, n$, and only $\mathbf{P}_i$ has a one at the position (i, i). These elements are from the first set of positions of the permutation matrices, as identified in (5.9). When put together, these n entries form a permutation matrix $\mathbf{P}^{(1)}$.

2. $a_{i,j} = 1$ for all $i = 1, \ldots, n$ and $j = ((i + 1) \bmod n) + 1$, and only $\mathbf{P}_h$, where $h = (i \bmod n) + 1$, has a one at the position (i, j). These elements are from the second set of positions of the permutation matrices, as identified in (5.9). When put together, these n entries form a permutation matrix $\mathbf{P}^{(2)}$.

3. $a_{i,j} = n - 2$ for all $i = 1, \ldots, n$ and $j = (i \bmod n) + 1$, where all $\mathbf{P}_\ell$ for $\ell \in \{1, \ldots, n\} \setminus \{i, j\}$ have a one at the position $a_{i,j}$. These elements are from the third set of positions of the permutation matrices, as identified in (5.9). When put together, these n entries form a permutation matrix $\mathbf{P}^{(3)}$ multiplied by the scalar $(n-2)$.

In other words, we can write

$$\mathbf{A} = \mathbf{P}^{(1)} + \mathbf{P}^{(2)} + (n-2) \cdot \mathbf{P}^{(3)} ,$$

and see that $\mathbf{A}$ has a BvN decomposition with three permutation matrices. We note that each row and column of $\mathbf{A}$ contains three nonzeros; and hence three is the smallest number of permutation matrices in a BvN decomposition of $\mathbf{A}$.

Since the minimum element in $\mathbf{A}$ is 1, and each $\mathbf{P}_i$ contains one such element, the Birkhoff heuristic can obtain a decomposition using $\mathbf{P}_i$ for $i = 1, \ldots, n$. Therefore, the Birkhoff heuristic's approximation is no better than $\frac{n}{3}$, which can be made arbitrarily large. □

We note that $\text{Greedy}_{\text{BvN}}$ will optimally decompose the matrix $\mathbf{A}$ used in the proof above.

We now analyze the theoretical properties of $\text{Greedy}_{\text{BvN}}$. We first give a lemma, identifying a pattern in its output.

Lemma 5.5. *The heuristic* $\text{Greedy}_{\text{BvN}}$ *obtains* $\alpha_1, \ldots, \alpha_k$ *in a non-increasing order* $\alpha_1 \geq \cdots \geq \alpha_k$*, where* α_j *is obtained at the* j*th step.*

We observed through a series of experiments that the performance of $\text{Greedy}_{\text{BvN}}$ depends on the values in the matrix, and there seems to be no constant factor approximation [74].

We create a set of $n \times n$ matrices. To do that, we first fix a set of z permutation matrices $\{\mathbf{C}_1, \ldots, \mathbf{C}_z\}$ of size $n \times n$. These permutation matrices with varying values of α will be used to generate the matrices. The matrices are parameterized by the subscript i and each $\mathbf{A}_i$ is created as follows: $\mathbf{A}_i = \alpha_1 \cdot \mathbf{C}_1 + \alpha_2 \cdot \mathbf{C}_2 + \cdots + \alpha_z \cdot \mathbf{C}_z$ where each α_j for $j = 1, \ldots, z$ is a randomly chosen integer in the range $[1, 2^i]$, and we also set a randomly chosen α_j equivalent to 2^i to guarantee the existence of at least one large value even in the unlikely case that all other values are not large enough. As can be seen, each $\mathbf{A}_i$ has the same structure and differs from the rest only in the values of α_j's that are chosen. As a consequence, they all can be decomposed by the same set of permutation matrices.

We present our results in two sets of experiments shown in Table 5.1, for two different n. We create five random $\mathbf{A}_i$ for $i \in \{10, 20, 30, 40, 50\}$, that is we have five matrices with the parameter i, and there are five different i. We have $n = 30$ and $z = 20$ in the first set, and $n = 200$ and $z = 100$ in the second set. Let k_i be the number of permutation matrices $\text{Greedy}_{\text{BvN}}$

$n = 30$ and $z = 20$				
	average		worst case	
i	k_i	k_i/z	k_i	k_i/z
10	59	2.99	63	3.15
20	105	5.29	110	5.50
30	149	7.46	158	7.90
40	184	9.23	191	9.55
50	212	10.62	227	11.35

$n = 200$ and $z = 100$				
	average		worst case	
i	k_i	k_i/z	k_i	k_i/z
10	268	2.69	280	2.80
20	487	4.88	499	4.99
30	716	7.16	726	7.26
40	932	9.33	947	9.47
50	1124	11.25	1162	11.62

Table 5.1: Experiments showing the dependence of the performance of $\text{Greedy}_{\text{BvN}}$ on the values of the matrix elements. n is the matrix size; $i \in \{10, 20, 30, 40, 50\}$ is the parameter for creating matrices using $\alpha_j \in [1, 2^i]$; z is the number of permutation matrices used in creating $\mathbf{A}_i$. Five experiments for a given pair of n and i. $\text{Greedy}_{\text{BvN}}$ obtains k_i permutation matrices. The average and the maximum number of permutation matrices obtained by $\text{Greedy}_{\text{BvN}}$ for five random instances are given. k_i/z is a lower bound on the performance of $\text{Greedy}_{\text{BvN}}$, as $z \geq Opt$.

obtains for a given $\mathbf{A}_i$. The table gives the average and the maximum k_i of five different $\mathbf{A}_i$, for a given n and i pair. By construction, each $\mathbf{A}_i$ has a BvN with z permutation matrices. Since z is no smaller than the optimal value, the ratio $\frac{k_i}{z}$ gives a lower bound on the performance of $\text{Greedy}_{\text{BvN}}$. As seen from the experiments, as i increases, the performance of $\text{Greedy}_{\text{BvN}}$ gets increasingly worse. This shows that a constant ratio worst case approximation of $\text{Greedy}_{\text{BvN}}$ is unlikely. While the performance depends on z (for example, for small z, $\text{Greedy}_{\text{BvN}}$ is likely to obtain near optimal decompositions), it seems that the size of the matrix does not largely affect the relative performance of $\text{Greedy}_{\text{BvN}}$.

Now we attempt to explain the above results theoretically.

Lemma 5.6. *Let $\alpha_1^\star \mathbf{P}_1^\star + \cdots + \alpha_k^\star \mathbf{P}_{k^\star}^\star$ be a BvN decomposition of a given doubly stochastic matrix $\mathbf{A}$ with the smallest number $k^\star$ of permutation matrices. Then, for any BvN decomposition of $\mathbf{A}$ with $\ell \geq k^\star$ permutation matrices, we have $\ell \leq k^\star \cdot \frac{\max_i \alpha_i^\star}{\min_i \alpha_i}$. If the coefficients are integers (e.g., when $\mathbf{A}$ is a matrix with constant row and column sums of integral values), we have $\ell \leq k^\star \cdot \max_i \alpha_i^\star$.*

Proof. Consider a BvN decomposition $\alpha_1 \mathbf{P}_1 + \cdots + \alpha_\ell \mathbf{P}_\ell$ with $\ell \geq k^\star$. Assume without loss of generality that $\alpha_1^\star \geq \cdots \geq \alpha_{k^\star}^\star$ and $\alpha_1 \geq \cdots \geq \alpha_\ell$.

We know that the coefficients of these two decompositions sum up to the same value. That is

$$\sum_{i=1}^{\ell} \alpha_i = \sum_{i=1}^{k^\star} \alpha_i^\star \, .$$

Since α_ℓ is the smallest of α, and $\alpha_1^\star$ is the largest of $\alpha^\star$, we have

$$\ell \cdot \alpha_\ell \leq k^\star \cdot \alpha_1^\star \,,$$

and hence

$$\frac{\ell}{k^\star} \leq \frac{\alpha_1^\star}{\alpha_\ell} \,.$$

By assuming integer values, we see that $\alpha_\ell \geq 1$ and thus

$$\ell \leq k^\star \cdot \max_i \alpha_i^\star \,.$$

□

This lemma evaluates the approximation guarantee of a given BvN decomposition. It does not seem very useful, because of the fact that even if we have $\min_i \alpha_i$, we do do not have $\max_i \alpha_i^\star$. Luckily, we can say more in the case of $\text{Greedy}_{\text{BvN}}$.

Corollary 5.1. *Let $k^\star$ be the smallest number of permutation matrices in a BvN decomposition of a given doubly stochastic matrix $\mathbf{A}$. Let α_1 and α_ℓ be the first and last coefficients obtained by the heuristic* $\text{Greedy}_{\text{BvN}}$ *for decomposing $\mathbf{A}$. Then, $\ell \leq k^\star \cdot \frac{\alpha_1}{\alpha_\ell}$.*

Proof. This is easy to see, as $\text{Greedy}_{\text{BvN}}$ obtains the coefficients in a non-increasing order (see Lemma 5.5), and $\alpha_1 \geq \alpha_j^\star$ for all $1 \leq j \leq k^\star$ for any BvN decomposition containing $\alpha_j^\star$. □

Lemma 5.6 and Corollary 5.1 give a posteriori estimates of the performance of the heuristic $\text{Greedy}_{\text{BvN}}$, in that one looks at the decomposition and tells how good it is. This potentially can reveal a good performance. For example, when $\text{Greedy}_{\text{BvN}}$ obtains a BvN decomposition with all coefficients equivalent, then we know that it is an optimal BvN. The same cannot be told for the Birkhoff heuristic though (consider the example proceeding Lemma 5.4). We also note that the ratio given in Corollary 5.1 should usually be much larger than the practical performance.

5.4 Experiments

We present results with the two heuristics. We note that the original Birkhoff heuristic was not concerned with the minimality of the number of permutation matrices. Therefore, the presented results are not for comparing the two heuristics, but for giving results with what is available. We give results on two different set of matrices. The first set contains real world, sparse matrices, preprocessed to be doubly stochastic. The second set contains a few randomly created, dense, doubly stochastic matrices.

					Birkhoff		$\text{Greedy}_{\text{BvN}}$	
matrix	n	τ	$d_{\max}$	dev.	$\sum_{i=1}^{k} \alpha_i$	k	$\sum_{i=1}^{k} \alpha_i$	k
aft01	8205	125567	21	0.0e+00	0.160	2000	1.000	120
aft02	8184	127762	82	1.0e-06	0.000	1434	1.000	234
barth	6691	46187	13	0.0e+00	0.160	2000	1.000	71
barth4	6019	40965	13	0.0e+00	0.140	2000	1.000	61
bcspwr10	5300	21842	14	1.0e-06	0.380	2000	1.000	63
bcsstk38	8032	355460	614	0.0e+00	0.000	2000	1.000	592∗
benzene	8219	242669	37	0.0e+00	0.000	2000	1.000	113
c-29	5033	43731	481	1.0e-06	0.000	2000	1.000	870
EX5	6545	295680	48	0.0e+00	0.020	2000	1.000	229
EX6	6545	295680	48	1.0e-06	0.030	2000	1.000	226
flowmeter0	9669	67391	11	0.0e+00	0.510	2000	1.000	58
fv1	9604	85264	9	0.0e+00	0.620	2000	1.000	50
fv2	9801	87025	9	0.0e+00	0.620	2000	1.000	52
fxm3_6	5026	94026	129	1.0e-06	0.130	2000	1.000	383
g3rmt3m3	5357	207695	48	1.0e-06	0.050	2000	1.000	223
Kuu	7102	340200	98	0.0e+00	0.000	2000	1.000	330
mplate	5962	142190	36	0.0e+00	0.030	2000	1.000	153
n3c6-b7	6435	51480	8	0.0e+00	1.000	8	1.000	8
nemeth02	9506	394808	52	0.0e+00	0.000	2000	1.000	109
nemeth03	9506	394808	52	0.0e+00	0.000	2000	1.000	115
olm5000	5000	19996	6	1.0e-06	0.750	283	1.000	14
s1rmq4m1	5489	262411	54	0.0e+00	0.000	2000	1.000	211
s2rmq4m1	5489	263351	54	0.0e+00	0.000	2000	1.000	208
SiH4	5041	171903	205	0.0e+00	0.000	2000	1.000	574
t2d_q4	9801	87025	9	2.0e-06	0.500	2000	1.000	54

Table 5.2: Birkhoff's heuristic and $\text{Greedy}_{\text{BvN}}$ on sparse matrices from the UFL collection. The column τ contains the number of nonzeros in a matrix. The column $d_{\max}$ contains the maximum number of nonzeros in a row or a column, setting up a lower bound for the number k of permutation matrices in a BvN decomposition. The column "dev." contains the maximum deviation of a row/column sum of a matrix $\mathbf{A}$ from 1, in other words the value $\max\{\|\mathbf{A1} - \mathbf{1}\|_\infty, \|\mathbf{1^T A} - \mathbf{1^T}\|_\infty\}$ reported to six significant digits. The two heuristics are run to obtain at most 2000 permutation matrices, or until they accumulated a sum of at least 0.9999 with the coefficients. In one matrix (marked with $*$), $\text{Greedy}_{\text{BvN}}$ obtained this number in less than $d_{\max}$ permutation matrices—increasing the limit to 0.999999 made it return with 908 permutation matrices.

The first set of matrices was created as follows. We have selected all matrices with the following properties from the SuiteSparse Matrix Collection (UFL) [59]: square, has at least 5000 and at most 10000 rows, fully indecomposable, and there are at most 50 and at least 2 nonzeros per row. This gave a set of 66 matrices. These 66 matrices are from 30 different groups of problems; we have chosen at most two per group to remove any bias that might be arising from the group. This resulted in 32 matrices which we preprocessed as follows to obtain a set of doubly stochastic matrices. We first took the absolute values of the entries to make the matrices nonnegative. Then, we scaled them to be doubly stochastic using the algorithm of Knight et al. [141] with a tolerance of 10^{-6} with at most 1000 iterations. With this setting, the scaling algorithm tries to get the maximum deviation of a row/column sum from 1 to be less than 10^{-6} within 1000 iterations. In 7 matrices, the deviation was larger than 10^{-4}. We deemed this too big a deviation from a doubly stochastic matrix and discarded those matrices. At the end, we had 25 matrices given in Table 5.2.

We run the two heuristics for the BvN decomposition to obtain at most 2000 permutation matrices, or until they accumulated a sum of at least 0.9999 with the coefficients. The accumulated sum of coefficients $\sum_{i=1}^{k} \alpha_i$ and the number of permutation matrices k for the Birkhoff and $\text{Greedy}_{\text{BvN}}$ heuristics are given in Table 5.2. As seen in this table, $\text{Greedy}_{\text{BvN}}$ obtains a much smaller number of permutation matrices than Birkhoff's heuristic, except the matrix `n3c6-b7`. This is a special matrix, with eight nonzeros in each row and column and all nonzeros are 1/8. Both heuristics find the same (minimum) number of permutation matrices. We note that the geometric mean of the ratios $k/d_{\max}$ is 3.4 for $\text{Greedy}_{\text{BvN}}$.

The second set of matrices was created as follows. For $n \in \{100, 200, 300\}$, we created five matrices of size $n \times n$ whose entries are randomly chosen integers between 1 and 100 (we performed tests with integers between 1 and 20 and the results were close to what we present here). We then scaled them to be doubly stochastic using the algorithm of Knight et al. [141] with a tolerance of 10^{-6} with at most 1000 iterations. We run the two heuristics for the BvN decomposition such that at most $n^2 - 2n + 2$ permutation matrices are found, or the total value of the coefficients was larger than 0.9999. We then took the average of the five instances with the same n. The results are in Table 5.3. In this table again, we see that $\text{Greedy}_{\text{BvN}}$ obtains much smaller k than Birkhoff's heuristic.

The experimental observation that $\text{Greedy}_{\text{BvN}}$ performs better than the Birkhoff heuristic is not surprising. This is for two reasons. First, as stated before, the original Birkhoff heuristic is not concerned with the number of permutation matrices. Second, the heuristic $\text{Greedy}_{\text{BvN}}$ is an adaptation of the Birkhoff heuristic to have large reductions at each step.

	Birkhoff		$\text{Greedy}_{\text{BvN}}$	
n	$\sum_{i=1}^{k} \alpha_i$	k	$\sum_{i=1}^{k} \alpha_i$	k
100	0.99	9644	1.00	388
200	0.99	39208	1.00	717
300	1.00	88759	1.00	1042

Table 5.3: Birkhoff's heuristic and $\text{Greedy}_{\text{BvN}}$ on random dense matrices. The maximum deviation of a row/column sum of a matrix $\mathbf{A}$ from 1, that is $\max\{\|\mathbf{A1} - \mathbf{1}\|_\infty, \|\mathbf{1^T A} - \mathbf{1^T}\|_\infty\}$ was always less than 10^{-6}. For each n, the results are the averages of five different instances.

5.5 A generalization of Birkhoff-von Neumann theorem

We have exploited the BvN decomposition in the context of preconditioning for solving linear systems [23]. As the decomposition is defined for nonnegative matrices, we needed a generalization of it to cover all real matrices. To motivate the generalized decomposition theorem, we give a brief summary of the preconditioner.

5.5.1 Doubly stochastic splitting

For a given nonnegative matrix $\mathbf{A}$, we first preprocess it to get a doubly stochastic matrix (whose row and column sums are one). Then using this doubly stochastic matrix, we select some fraction of some of the nonzeros of $\mathbf{A}$ to be included in the preconditioner. The selection is done by taking a set of permutation matrices along with their coefficients in a BvN decomposition of the given matrix.

Assume we have a BvN decomposition of $\mathbf{A}$ as shown in (5.1). Then, one can pick an integer r between 1 and $k-1$ and split $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{M} - \mathbf{N}, \tag{5.10}$$

where

$$\mathbf{M} = \alpha_1 \mathbf{P}_1 + \cdots + \alpha_r \mathbf{P}_r, \quad \mathbf{N} = -\alpha_{r+1} \mathbf{P}_{r+1} - \cdots - \alpha_k \mathbf{P}_k. \tag{5.11}$$

Note that M and $-N$ are doubly substochastic matrices.

Definition 5.1. *A splitting of the form* (5.10) *with* $\mathbf{M}$ *and* $\mathbf{N}$ *given by* (5.11) *is said to be a* doubly substochastic splitting.

Definition 5.2. *A doubly substochastic splitting* $\mathbf{A} = \mathbf{M} - \mathbf{N}$ *of a doubly stochastic matrix* $\mathbf{A}$ *is said to be* standard *if* $\mathbf{M}$ *is invertible. We will call such a splitting an* SDS splitting.

In general, it is not easy to guarantee that a given doubly substochastic splitting is standard, except for some trivial situation such as the case $r = 1$, in which case $\mathbf{M}$ is always invertible. We also have a characterization for invertible $\mathbf{M}$ when $r = 2$.

Theorem 5.2. *A sufficient condition for $M = \sum_{i=1}^{r} \alpha_i P_i$ to be invertible is that one of the α_i with $1 \leq i \leq r$ be greater than the sum of the remaining ones.*

Let $\mathbf{A} = \mathbf{M} - \mathbf{N}$ be an SDS splitting of $\mathbf{A}$ and consider the stationary iterative scheme

$$x^{k+1} = \mathbf{H}x^k + c, \quad \mathbf{H} = \mathbf{M}^{-1}\mathbf{N}, \quad c = \mathbf{M}^{-1}b\,, \tag{5.12}$$

where $k = 0, 1, \ldots$ and x^0 is arbitrary. As seen in (5.12), $\mathbf{M}^{-1}$ or solvers for $\mathbf{M}y = z$ are required. As is well known, the scheme (5.12) converges to the solution of $\mathbf{A}x = b$ for any x^0 if and only if $\rho(\mathbf{H}) < 1$. Hence, we are interested in conditions that guarantee that the spectral radius of the iteration matrix

$$\mathbf{H} = \mathbf{M}^{-1}\mathbf{N} = -(\alpha_1\mathbf{P}_1 + \cdots + \alpha_r\mathbf{P}_r)^{-1}(\alpha_{r+1}\mathbf{P}_{r+1} + \cdots + \alpha_k\mathbf{P}_k)$$

is strictly less than one. In general, this problem appears to be difficult. We have a necessary condition (Theorem 5.3), and a sufficient condition (Theorem 5.4) which is simple but restrictive.

Theorem 5.3. *For the splitting $\mathbf{A} = \mathbf{M} - \mathbf{N}$ with $\mathbf{M} = \sum_{i=1}^{r} \alpha_i \mathbf{P}_i$ and $\mathbf{N} = -\sum_{i=r+1}^{k} \alpha_i \mathbf{P}_i$ to be convergent, it must hold that $\sum_{i=1}^{r} \alpha_i > \sum_{i=r+1}^{k} \alpha_i$.*

Theorem 5.4. *Suppose that one of the α_i appearing in $\mathbf{M}$ is greater than the sum of all the other α_i. Then $\rho(\mathbf{M}^{-1}\mathbf{N}) < 1$ and the stationary iterative method (5.12) converges for all x^0 to the unique solution of $\mathbf{A}x = b$.*

We discuss how to build preconditioners meeting the sufficiency conditions. Since $\mathbf{M}$ itself is doubly stochastic (up to a scalar ratio), we can apply splitting recursively on $\mathbf{M}$ and obtain a special solver. Our motivation is that the preconditioner $\mathbf{M}^{-1}$ can be applied to vectors via a number of highly concurrent steps, where the number of steps is controlled by the user. Therefore, the preconditioners (or the splittings) can be advantageous for use in many-core computing systems. In the context of splittings, the application of $\mathbf{N}$ to vectors also enjoys the same property. These motivations are shared by recent work on ILU preconditioners, where their fine-grained computation [48] and approximate application [12] are investigated for GPU-like systems.

Preconditioner construction

It is desirable to have a small number k in the Birkhoff-von Neumann decomposition while designing the preconditioner—this was our motivation to investigate the MinBvNDec problem. This is because of the fact that if we use splitting, then k determines the number of steps in which we compute the matrix-vector products. If we do not use all k permutation matrices, having a few with large coefficients should help to design the preconditioner. As stated in Lemma 5.5, $\text{Greedy}_{\text{BvN}}$ (Algorithm 11) obtains the coefficients in a non-increasing order. We can therefore use $\text{Greedy}_{\text{BvN}}$ to build an $\mathbf{M}$ such that it satisfies the sufficiency condition presented in Theorem 5.4. That is, we can have $\mathbf{M}$ with $\frac{\alpha_1}{\sum_{i=1}^{k}\alpha_i} > 1/2$, and hence $\mathbf{M}^{-1}$ can be applied with splitting iterations. For this, we start by initializing $\mathbf{M}$ to $\alpha_1\mathbf{P}_1$. Then, when a coefficient α_j where $j \geq 2$ is obtained at Line 5 of Algorithm 11, we check if α_1 will still be larger than the sum of other coefficients used in building $\mathbf{M}$, when we add $\alpha_j\mathbf{P}_j$ to $\mathbf{M}$. If so, we add $\alpha_j\mathbf{P}_j$ to $\mathbf{M}$ and continue. In practice, we iterate the while loop until k is around 10 and collect α_j's as described above. Experiments on a set of challenging problems [23] show that this preconditioner is more effective than ILU(0).

5.5.2 Birkhoff von-Neumann decomposition for arbitrary matrices

In order to apply the preconditioners to all fully indecomposable sparse matrices, we generalized the BvN decomposition as follows.

Theorem 5.5. *Any (real) matrix* $\mathbf{A}$ *with total support can be written as a convex combination of a set of signed, scaled permutation matrices.*

Proof. Let $\mathbf{B} = \text{abs}(\mathbf{A})$ and consider $\mathbf{R}$ and $\mathbf{C}$ making $\mathbf{RBC}$ doubly stochastic, which has a Birkhoff-von Neumann decomposition $\mathbf{RBC} = \sum_{i=1}^{k}\alpha_i\mathbf{P}_i$. Then, $\mathbf{RAC}$ can be expressed as

$$\mathbf{RAC} = \sum_{i=1}^{k}\alpha_i\mathbf{Q}_i \ ,$$

where $\mathbf{Q}_i = [q_{jk}^{(i)}]_{n\times n}$ is obtained from $\mathbf{P}_i = [p_{jk}^{(i)}]_{n\times n}$ as follows:

$$q_{jk}^{(i)} = \text{sign}(a_{jk})p_{jk}^{(i)} \ .$$

The scaling matrices can be inverted to express $\mathbf{A}$. □

De Werra [62] presents another generalization of the Birkhoff-von Neumann theorem to arbitrary matrices. In De Werra's generalization, instead of permutation matrices, signed integral matrices are used to decompose the

given matrix. The entries of the integral matrices used in the decomposition have the same sign as the corresponding entries in the original matrix, but there may be multiple nonzeros in a row or column. De Werra discusses flow-based algorithms to obtain the decompositions. While the decomposition is costlier to obtain (general flow computations are usually costlier than computing perfect matchings), it is also more general to handle rectangular matrices as well.

5.6 Summary, further notes and references

We investigated the problem of obtaining a Birkhoff-von Neumann decomposition of doubly stochastic matrices with the minimum number of permutation matrices. We showed four results regarding this problem. First, obtaining a BvN decomposition with the minimum number of permutation matrices is NP-hard. Second, there are matrices whose decompositions with the smallest number of permutation matrices cannot be obtained by any Birkhoff-like heuristic. Third, the worst-case approximation ratio of the original Birkhoff heuristic is $\Omega(n)$. On a more practical side, we proposed a natural greedy heuristic ($\text{Greedy}_{\text{BvN}}$) and presented experimental results which demonstrated that this heuristic delivers results that are not far from a trivial lower bound. We theoretically investigated $\text{Greedy}_{\text{BvN}}$'s performance and observed that its performance depends on the values of matrix elements. We showed a bound using the first and the last coefficients found by the heuristic. This chapter also included a generalization of the Birkhoff-von Neumann theorem, in which we are able to express any real matrix with a total support as a convex combination of scaled, signed permutation matrices.

The shown bound for the performance of $\text{Greedy}_{\text{BvN}}$ is expected to be much larger than what one observes in practice, as the bound can even be larger than the upper bound on the number of permutation matrices. A tighter analysis should be possible to explain the practical performance of the $\text{Greedy}_{\text{BvN}}$ heuristic.

The heuristics we considered in this chapter were based on finding a perfect matching. Koopmans and Beckmann [143] present another constructive proof of the Birkhoff theorem. This constructive proof splits the matrix as $\mathbf{A} = \beta\mathbf{A}_1 + (1-\beta)\mathbf{A}_2$, where $0 < \beta < 1$, $\mathbf{A}_1$ and $\mathbf{A}_2$ being doubly stochastic matrices. A related proof is also given by Horn and Johnson [118, Theorem 8.7.1], where $\beta = 1/2$. We review these two constructive proofs.

Let C be a cycle of length 2ℓ with ℓ vertices in each part of the bipartite graph, with vertices $r_i, r_{i+1}, \ldots, r_{i+\ell-1}$ on one side and $c_i, c_{i+1}, \ldots, c_{i+\ell-1}$ on the other side. Let us imagine the cycle drawn on the plane so that we have horizontal edges of the form (r_j, c_j) for $j = i, \ldots, i+\ell-1$, slanted edges of the form $(r_i, c_{i+\ell-1})$ and (r_j, c_{j-1}) for $j = i+1, \ldots, i+\ell-1$. A

(a) $\varepsilon_1 = \min\{a_{ii}, a_{jj}\}$ (b) $\varepsilon_2 = \min\{a_{ij}, a_{ji}\}$

Figure 5.4: A small example for Koopmans and Beckmann decomposition approach.

small example is shown in Figure 5.4. Let ε_1 and ε_2 be the minimum value of a horizontal and slanted edge, respectively. Koopmans and Beckmann proceed as follows to obtain a BvN decomposition. Consider a matrix $\mathbf{C}_1$ which contains $-\varepsilon_1$ at the positions corresponding to the horizontal edges of C, ε_1 at the positions corresponding to the slanted edges of C, and zeros elsewhere. Then, define $\mathbf{A}_1 = \mathbf{A} + \mathbf{C}_1$ (see Fig. 5.4a). It is easy to see that each element of $\mathbf{A}_1$ is between 0 and 1, and row and column sums are the same as in $\mathbf{A}$. Therefore $\mathbf{A}_1$ is doubly stochastic. Similarly, consider $\mathbf{C}_2$ which contains ε_2 at the positions corresponding to the horizontal edges of C, $-\varepsilon_2$ at the positions corresponding to the slanted edges of C, and zeros elsewhere. Then, define $\mathbf{A}_2 = \mathbf{A} + \mathbf{C}_2$ (see Fig. 5.4b) and observe that $\mathbf{A}_2$ is also doubly stochastic. By a simple arithmetic, we can set $\beta = \frac{\varepsilon_2}{\varepsilon_1+\varepsilon_2}$ and write $\mathbf{A} = \beta\mathbf{A}_1 + (1-\beta)\mathbf{A}_2$. The observation that $\mathbf{A}_1$ and $\mathbf{A}_2$ contain at least one more zero than $\mathbf{A}$ can then be used in an inductive hypothesis to construct a decomposition, by recursively decomposing $\mathbf{A}_1$ and $\mathbf{A}_2$.

At first sight, the constructive approach of Koopmans and Beckmann does not lead to an efficient algorithm. This is because of the fact that $\mathbf{A}_1$ and $\mathbf{A}_2$ can have significant overlap and hence many permutation matrices can be shared by $\mathbf{A}_1$ and $\mathbf{A}_2$, yielding combinatorially large space requirement or very high run time. A variant of this algorithm needs to be explored. First, note that we can find a cycle and manipulate the weights by decreasing them along the horizontal edges and increasing them along the slanted edges (not necessarily by annihilating the smallest weight) to create another matrix. Then, we can take the modified matrix and manipulate another cycle. At one point, one can decide to stop and create $\mathbf{A}_1$ this way; with a suitable choice of coefficients the decomposition can thus proceed. At the second sight, then, one observes that Koopmans and Beckmann method calls for creative ideas for developing efficient and effective heuristics for BvN decompositions.

Horn and Johnson, on the other hand, set ε to the smallest of ε_1 and ε_2.

Without loss of generality, let ε_1 be the smallest. Then, let $\mathbf{C}$ be a matrix whose entries are $\{-1, 0, 1\}$ corresponding to the negative, zero, and positive entries of $\mathbf{C}_1$. Then let $\mathbf{A}_1 = \mathbf{A} + \varepsilon\mathbf{C}$ and $\mathbf{A}_2 = \mathbf{A} - \varepsilon\mathbf{C}$. Both of these matrices have nonnegative entries less than or equal to one, and have the same row and column sums as $\mathbf{A}$. We can then write $\mathbf{A} = 0.5\mathbf{A}_1 + 0.5\mathbf{A}_2$ and recurse on $\mathbf{A}_1$ and $\mathbf{A}_2$ to obtain a decomposition (which stops when we do not have any cycles).

Inspired by the approaches summarized above, we can optimally decompose the matrix (5.8) used in showing that there are optimal decompositions which cannot be obtained by Birkhoff-like heuristics. Let

$$\mathbf{A}_1 = \begin{pmatrix} a+b & d & c & e & 0 \\ e & a+c & b & d & 0 \\ 0 & e & d & b+c & a \\ d & b & a & 0 & c+e \\ c & 0 & e & a & b+d \end{pmatrix},$$

$$\mathbf{A}_2 = \begin{pmatrix} a+b & d+2\cdot i & c+2\cdot h & e+2\cdot j & 2\cdot f+2\cdot g \\ e+2\cdot g & a+c & b+2\cdot i & d+2\cdot f & 2\cdot h+2\cdot j \\ 2\cdot f+2\cdot j & e+2\cdot h & d+2\cdot g & b+c & a+2\cdot i \\ d+2\cdot h & b+2\cdot f & a+2\cdot j & 2\cdot g+2\cdot i & c+e \\ c+2\cdot i & 2\cdot g+2\cdot j & e+2\cdot f & a+2\cdot h & b+d \end{pmatrix},$$

and observe that $\mathbf{A} = 0.5\mathbf{A}_1 + 0.5\mathbf{A}_2$. We can then use the permutations containing the coefficients a, b, c, d, e to decompose $\mathbf{A}_1$ optimally with five permutation matrices using $\text{Greedy}_{\text{BvN}}$ (in the decreasing order: first e, then d, then c and so on). While decomposing $\mathbf{A}_2$ we need to use the permutation matrices found for $\mathbf{A}_1$ with a careful selection of the coefficients (instead of $a+b$, we use a for example for the permutation associated with a). Once we have consumed those five permutations, we can again resort to $\text{Greedy}_{\text{BvN}}$ to decompose the remaining matrix optimally with the five permutation matrices associated with f, g, h, i, and j. In this way, we have an optimal decomposition for the matrix (5.8), by applying $\text{Greedy}_{\text{BvN}}$ to $\mathbf{A}_1$ and carefully decomposing $\mathbf{A}_2$. We note that neither $\mathbf{A}_1$ nor $\mathbf{A}_2$ have the same sum as $\mathbf{A}$, in contrast to the two approaches above. Can we systematize this line of reasoning to develop an effective heuristic to obtain a BvN decomposition?

Chapter 6

Matchings in hypergraphs

In this chapter, we investigate the maximum matching in d-partite, d-uniform hypergraphs described in Section 1.3. The formal problem definition is repeated below for convenience.

Maximum matching in d-partite d-uniform hypergraphs. Given a d-partite d-uniform hypergraph, find a matching of maximum size.

We investigate effective heuristics for the problem above. This problem has been studied mostly in the context of local search algorithms [119], and the best known algorithm is due to Cygan [56] who provides a $((d+1+\varepsilon)/3)$-approximation, building on previous work [57, 106]. It is NP-hard to approximate the 3-partite case (Max-3-DM) within 98/97 [24]. Similar bounds exist for higher dimensions: the hardness of approximation for $d = 4, 5$ and 6 are shown to be $54/53 - \varepsilon$, $30/29 - \varepsilon$, and $23/22 - \varepsilon$, respectively [109].

We propose five heuristics for the above problem. The first two heuristics are the adaptations of the Greedy [78] and Karp-Sipser [128] heuristics used in finding matchings in graphs (they are summarized in Chapter 4). The proposed generalizations are referred to as Greedy-H and Karp-Sipser-H. Greedy-H traverses the hyperedge list in random order and adds an edge to the matching whenever possible. Karp-Sipser-H introduces certain rules to Greedy-H to improve the cardinality. The third heuristic is inspired by a recent scaling-based approach proposed for the maximum cardinality matching problem on graphs [73, 74, 76]. The fourth heuristic is a modification on the third one that allows for faster execution time. The last one finds a matching for a reduced, $(d-1)$-dimensional problem and exploits it for the original matching problem recursively. This heuristic uses an exact algorithm for the bipartite matching problem at each reduction. We perform experiments to evaluate the performance of these heuristics on special classes of random hypergraphs as well as real-life data.

One plausible way to tackle the problem is to create the line graph G for a given hypergraph H. The line graph is created by identifying each

hyperedge of H with a vertex in G, and by connecting two vertices of G with an edge, iff the corresponding hyperedges share a common vertex in H. Then, successful heuristics for computing large independent sets in graphs, e.g., KaMIS [147], can be used to compute large matchings in hypergraphs. This approach, although promising quality-wise, could be impractical. This is so, since building G from H requires quadratic run time (in terms of the number of hyperedges) and more importantly quadratic storage (again in terms of the number of hyperedges) in the worst case. While this can be acceptable in some instances, in some others it is not. We have such instances in the experiments. Notice that while a heuristic for the independent set problem can be of linear time complexity in graphs, due to our graphs being a line graph, the actual complexity could be high.

6.1 Background and notation

Franklin and Lorenz [89] show that if a nonnegative d-dimensional tensor $\mathcal{X}$ has the same zero-pattern as a d-stochastic tensor, one can apply a multidimensional version of the Sinkhorn-Knopp algorithm [202] (see also the description in Section 4.1) to scale $\mathcal{X}$ to be d-stochastic. This is accomplished by finding d vectors $u^{(1)}, \ldots, u^{(d)}$ and scaling the nonzeros of $\mathcal{X}$ as $x_{i_1,\ldots,i_d} \cdot u^{(1)}_{i_1} \cdot \cdots \cdot u^{(d)}_{i_d}$ for all $i_1, \ldots, i_d \in \{1, \ldots, n\}$.

In a hypergraph, if a vertex is a member of only a single hyperedge, we call it a degree-1 vertex. Similarly, if a vertex is a member of only two hyperedges, we call it a degree-2 vertex.

In the *k-out random hypergraph model*, given a set V of vertices, each vertex $u \in V$ selects k hyperedges from the set $E_u = \{e : e \subseteq V, u \in e\}$ in a uniformly random fashion and the union of these edges forms E. We are interested in the d-partite, d-uniform case, and hence $E_u = \{e : |e \cap V_i| = 1 \text{ for } 1 \leq i \leq d, u \in e\}$. This model generalizes the random k-out bipartite graphs [211]. Devlin and Kahn [67] investigate fractional matchings in these hypergraphs, and mention in passing that k should be exponential in d to ensure that a perfect matching exists.

6.2 The heuristics

A matching which cannot be extended with more edges is called *maximal*. The heuristics proposed here find maximal matchings on d-partite, d-uniform hypergraphs. For such hypergraphs, any maximal matching is a d-approximate matching. The bound is tight and can be verified for $d = 3$. Let H be a 3-partite $3 \times 3 \times 3$ hypergraph with the following edges $e_1 = (1, 1, 1), e_2 = (2, 2, 2), e_3 = (3, 3, 3)$ and $e_4 = (1, 2, 3)$. The maximum matching is $\{e_1, e_2, e_3\}$ but the edge $\{e_4\}$ alone forms a maximal matching.

6.2.1 Greedy-H: A Greedy heuristic

Among the two variants of Greedy [78, 194], we adapt the first one, which randomly visits the edges. The adapted version is referred to as Greedy-H, and it visits the hyperedges in random order and adds the visited hyperedge to the matching whenever possible. Since only maximal matchings are possible as its output, Greedy-H is a d-approximation heuristic.

6.2.2 Karp-Sipser-H: Adapting Karp-Sipser

The commonly used and analyzed version of the Karp-Sipser heuristics for graphs applies degree-1 reductions, as summarized in Section 4.2.1. The original version though applies two rules:

- At any time during the heuristic, if a degree-1 vertex appears it is matched with its only neighbor.
- Otherwise, if a degree-2 vertex u appears with neighbors $\{v, w\}$, u (and its edges) is removed from the current graph, and v and w are merged to create a new vertex vw whose set of neighbors is the union of those of v and w (except u). A maximum cardinality matching for the reduced graph can be extended to obtain one for the current graph by matching u with either v or w depending on vw's match.

We now propose an adaptation of Karp-Sipser with the two rules for d-partite, d-uniform hypergraphs. The adapted version is called Karp-Sipser-H. Similar to the original one, the modified heuristic iteratively adds a random hyperedge to the matching, remove its d endpoints, as well as their hyperedges. However, the random selection is not applied whenever hyperedges defined by the following lemmas appear.

Lemma 6.1. *During the heuristic, if a hyperedge e with at least $d-1$ degree-1 endpoints appears, there exists a maximum cardinality matching in the current hypergraph containing e.*

Proof. Let H' be the current hypergraph at hand and $e = (u_1, \ldots, u_d)$ be a hyperedge in H' whose first $d-1$ endpoints are degree-1 vertices. Let M' be a maximum cardinality matching in H'. If $e \in M'$, we are done. Otherwise, assume that u_d is the endpoint matched by a hyperedge $e' \in M'$ (note that if u_d is not matched M' can be extended with e). Since u_i, $1 \leq i < d$, are not matched in M', $M' \setminus \{e'\} \cup \{e\}$ defines a valid maximum cardinality matching for H'. □

We note that it is not possible to relax the condition by using a hyperedge e with less than $d-1$ endpoints of degree-1; in M', two of e's higher degree endpoints could be matched with two different hyperedges, in which case the substitution as done in the proof of the lemma will not be valid.

Lemma 6.2. *During the heuristic, let $e = (u_1, \ldots, u_d)$ and $e' = (u'_1, \ldots, u'_d)$ be two hyperedges sharing at least one endpoint where for an index set $\mathcal{I} \subset \{1, \ldots, d\}$ of cardinality $d-1$, the vertices u_i, u'_i for all $i \in \mathcal{I}$ only touch e and/or e'. That is for each $i \in \mathcal{I}$, either $u_i = u'_i$ is a degree-2 vertex or $u_i \neq u'_i$ and they are both degree-1 vertices. For $j \notin \mathcal{I}, u_j$ and u'_j are arbitrary vertices. Then, in the current hypergraph, there exists a maximum cardinality matching having either e or e'.*

Proof. Let H' be the current hypergraph at hand and $j \notin \mathcal{I}$ be the remaining part id. Let M' be a maximum cardinality matching in H'. If either $e \in M'$ or $e' \in M'$, we are done. Otherwise, u_i and u'_i for all $i \in \mathcal{I}$ are unmatched by M'. Furthermore, since M' is maximal, u_j must be matched by M' (otherwise, M' can be extended by e). Let $e'' \in M'$ be the hyperedge matching u_j. Then $M' \setminus \{e''\} \cup \{e\}$ defines a valid maximum cardinality matching for H'. □

Whenever such hyperedges appear, the rules below are applied in the same order:

- **Rule-1**: At any time during the heuristic, if a hyperedge e with at least $d-1$ degree-1 endpoints appears, instead of a random edge, e is added to the matching and removed from the hypergraph.
- **Rule-2**: Otherwise, if two hyperedges e and e' as defined in Lemma 6.2 appear, they are removed from the current hypergraph with the endpoints u_i, u'_i for all $i \in \mathcal{I}$. Then, we consider u_j and u'_j. If u_j and u'_j are distinct, they are merged to create a new vertex $u_j u'_j$, whose hyperedge list is defined as the union of u_j's and u'_j's hyperedge lists. If u_j and u'_j are identical, we rename u_j as $u_j u'_j$. After obtaining a maximal matching on the reduced hypergraph, depending on the hyperedge matching $u_j u'_j$, either e or e' can be used to obtain a larger matching in the current hypergraph.

When Rule-2 is applied, the two hyperedges identified in Lemma 6.2 are removed from the hypergraph, and only the hyperedges containing u_j and/or u'_j have an update in their vertex list. Since the original hypergraph is d-partite and d-uniform, that update is just a renaming of a vertex in the concerned hyperedges (hence the resulting hypergraph is d-partite and d-uniform).

Although the extended rules usually lead to improved results in comparison to Greedy-H, Karp-Sipser-H still adheres to the d-approximation bound of maximal matchings. For the example given at the beginning of Section 6.2, Karp-Sipser-H generates a maximum cardinality matching by applying the first rule. However, when $e_5 = (2, 1, 3)$ and $e_6 = (3, 1, 3)$ are added to the example, neither of the two rules can be applied. As before, in case e_4 is randomly selected, it alone forms a maximal matching.

6.2.3 Karp-Sipser-H-scaling: Karp-Sipser with scaling

Karp-Sipser-H can be modified for better decisions in case neither of the two rules apply. In this variant, instead of a random selection, we first scale the adjacency tensor of H and obtain an approximate d-stochastic tensor $\mathcal{X}$. We then augment the matching by adding the edge which corresponds to the largest value in $\mathcal{X}$, when the rules do not apply. The modified heuristic is summarized in Algorithm 12.

Algorithm 12: Karp-Sipser-H-scaling (H)

Data: A d-partite d-uniform $n_1 \times \cdots \times n_d$ hypergraph $H = (V, E)$
Result: A maximal matching M of H

$M \leftarrow \emptyset$ /* Initially M is empty */
$S \leftarrow \emptyset$ /* Stack for the merges for Rule-2 */
while H *is not empty* **do**
  remove isolated vertices from H
  if $\exists e = (u_1, \ldots, u_d)$ *as in Rule-1* **then**
    $M \leftarrow M \cup \{e\}$ /* Add e to the matching */
    Apply the reduction Rule-1 on H
  else if $\exists e = (u_1, \ldots, u_d), e' = (u'_1, \ldots, u'_d)$ *and* $\mathcal{I}$ *as in Rule-2* **then**
    Let j be the part index where $j \notin \mathcal{I}$
    Apply the reduction Rule-2 on H by introducing the vertex $u_j u'_j$
    $E' = \{(v_1, \ldots, u_j u'_j, \ldots, v_d) : \text{for all } (v_1, \ldots, u_j, \ldots, v_d) \in E\}$
      /* memorize the hyperedges of u_j */
    S.push(e, e', $u_j u'_j$, E') /* Store the current merge */
  else
    $\mathcal{X} \leftarrow \textsc{Scale}(adj(H))$ /* Scale the adjacency tensor of H */
    $e \leftarrow \arg\max_{(u_1,\ldots,u_d)} (x_{u_1,\ldots,u_d})$ /* Find the max. in $\mathcal{X}$ */
    $M \leftarrow M \cup \{e\}$ /* Add e to the matching */
    Remove all hyperedges of $u_1, \ldots, u_d$ from E
    $V \leftarrow V \setminus \{u_1, \ldots, u_d\}$
while $S \neq \emptyset$ **do**
  $\langle e, e', u_j u'_j, E' \rangle \leftarrow S$.pop()
  if $u_j u'_j$ *is not matched by* M **then**
    $M \leftarrow M \cup \{e\}$
  else
    Let $e'' \in M$ be the hyperedge matching $u_j u'_j$
    if $e'' \in E'$ **then**
      Replace $u_j u'_j$ in e'' with u'_j
      $M \leftarrow M \cup \{e'\}$
    else
      Replace $u_j u'_j$ in e'' with u_j
      $M \leftarrow M \cup \{e\}$

Our inspiration comes from the $d = 2$ case [74, 76] described in Chapter 4. By using the scaling method as a preprocessing step and choosing

edges with a probability corresponding to the scaled entry, the edges which are not included in a perfect matching become less likely to be chosen. Unfortunately for $d \geq 3$, there is no equivalent of Birkhoff's theorem as demonstrated by the following lemma, whose proof can be found in the associated technical report [75, Lemma 3].

Lemma 6.3. *For $d \geq 3$, there exist extreme points in the set of d-stochastic tensors which are not permutation tensors.*

These extreme points can be used to generate other d-stochastic tensors as linear combinations. Due to the lemma above, we do not have the theoretical foundation to imply that hyperedges corresponding to the large entries in the scaled tensor must necessarily participate in a perfect matching. Nonetheless, the entries not in any perfect matching tend to become zero (not guaranteed for all though). For the worst case example of Karp-Sipser-H described above, the scaling indeed helps the entries corresponding to e_4, e_5 and e_6 to become zero. See the discussion following the said lemma in the technical report [75].

On a d-partite, d-uniform hypergraph $H = (V, E)$, the Sinkhorn-Knopp algorithm used for scaling operates in iterations, each of which requires $\mathcal{O}(|E| \times d)$ time. In practice, we perform only a few iterations (e.g., 10–20). Since, we can match at most $|V|/d$ hyperedges, the overall run time cost associated with scaling is $\mathcal{O}(|V| \times |E|)$. A straightforward implementation of the second rule can take quadratic time in the worst case of a large number of repetitive merges with a given vertex. In practice, more of a linear time behavior should be observed for the second rule.

6.2.4 Karp-Sipser-H-mindegree: Hypergraph matching via pseudo scaling

In Algorithm 12, applying scaling at every step can be very costly. Here we propose an alternative idea inspired by the specifics of the Sinkhorn-Knopp algorithm to reduce the overall cost. In particular, we can mimic the first iteration of the Sinkhorn-Knopp algorithm to discover that we can use the inverse of the degree of a vertex as its scaling vector. This assigns each vertex a weight proportional to the inverse of its degree—hence not an exact iteration of Sinkhorn-Knopp.

With this approach each hyperedge $\{i_1, \ldots, i_d\}$ is associated with a value $\frac{1}{\prod_{j=1}^{d} \lambda_{i_j}}$, where λ_{i_j} denotes the degree of the vertex i_j from jth part. The selection procedure is the same as that of Algorithm 12, i.e., the edge with the maximum value is added to the matching set. We refer to this algorithm as Karp-Sipser-H-mindegree, as it selects a hyperedge based on a function of the degrees of the vertices. With a straightforward implementation, finding this hyperedge takes $O(|E|)$ time. For a better efficiency, the edges can be

stored in a heap and when the degree of a node v decreases, the *increaseKey* heap operation can be called for all its edges.

6.2.5 Bipartite-reduction: Reduction to the bipartite matching problem

A perfect matching in a d-partite, d-uniform hypergraph H remains perfect when projected on a $(d-1)$-partite, $(d-1)$-uniform hypergraph obtained by removing one of H's dimensions. Matchability in $(d-1)$-dimensional sub-hypergraphs has been investigated [6] to provide an equivalent of Hall's Theorem for d-partite hypergraphs. These observations lead us to propose a heuristic called Bipartite-reduction. This heuristic tackles the d-partite, d-uniform case by recursively asking for matchings in $(d-1)$-partite, $(d-1)$-uniform hypergraphs and so on, until d=2.

Let us start with the case where $d = 3$. Let $G = (V_G, E_G)$ be the bipartite graph with the vertex set $V_G = V_1 \cup V_2$ obtained by deleting V_3 from a 3-partite, 3-regular hypergraph $H = (V, E)$. The edge $(u, v) \in E_G$ iff there exists a hyperedge $(u, v, z) \in E$. One can also assign a weight function $w(\cdot)$ to the edges during this step such as

$$w(u, v) = |\{z : (u, v, z) \in E\}| \,. \tag{6.1}$$

A maximum weighted (product, sum, etc.) matching algorithm can be used to obtain a matching M_G on G. A second bipartite graph $G' = (V_{G'}, E_{G'})$ is then created with $V_{G'} = (V_1 \times V_2) \cup V_3$ and $E_{G'} = \{(uv, z) : (u, v) \in M_G, (u, v, z) \in H\}$. Under this construction, any matching in G' corresponds to a valid matching in H. Furthermore, if the weight function (6.1) defined above is used, the following holds (the proof is in the technical report [75, Proposition 4]).

Proposition 6.1. *Let $w(M_G) = \sum_{(u,v)\in M_G} w(u, v)$ be the size of the matching M_G found in G. Then G' has $w(M_G)$ edges.*

Thus, by selecting a maximum weighted matching M_G and maximizing $w(M_G)$, the largest number of edges will be kept in G'.

For d-dimensional matching, a similar process is followed. First, an ordering $i_1, i_2, \ldots, i_d$ of the dimensions is defined. At the jth bipartite reduction step, the matching is found between the dimension cluster $i_1 i_2 \cdots i_j$ and dimension i_{j+1} by similarly solving a bipartite matching instance where the edge $(u_1 \cdots u_j, v)$ exists iff vertices $u_1, \ldots, u_j$ were matched in previous steps, and there exists an edge $(u_1, \ldots, u_j, v, z_{j+2}, \ldots, z_d)$ in H.

Unlike the previous heuristics, Bipartite-reduction does not have any approximation guarantee. We state this with the following lemma (the proof is in the technical report [75, Lemma 5]).

Lemma 6.4. *If algorithms for the maximum cardinality or the maximum weighted matching, with the suggested edge weights (6.1), problems are used, then* Bipartite-reduction *has a worst-case approximation ratio of* $\Omega(n)$.

6.2.6 Local-Search: Performing local search

A local search heuristic is proposed by Hurkens and Schrijver [119]. It starts from a feasible maximal matching M and performs a series of swaps until it is no longer possible. In a swap, k edges of M are replaced with at least $k+1$ new edges from $E \setminus M$ so that the cardinality of M increases by at least one. These k edges from M can be replaced with at most $d \times k$ new edges. Hence, these edges can be found by a polynomial algorithm enumerating all the possibilities. The approximation guarantee improves with higher k values. Local search algorithms are limited in practice due to their high time complexity. The algorithm might have to examine all $\binom{|M|}{k}$ subsets of M to find a feasible swap at each step. The algorithm by Cygan [56] which achieves a $\left(\frac{d+1+\varepsilon}{3}\right)$-approximation is based on a different swap scheme but is also not suited for large hypergraphs. We implement Local-Search as an alternative from the literature to set a baseline.

6.3 Experiments

To understand the relative performance of the proposed heuristics, we conducted a wide variety of experiments with both synthetic and real-life data. The experiments were performed on a computer equipped with Intel Core i7-7600 CPU and 16GB RAM. We investigate the performance of the proposed heuristics Greedy-H, Karp-Sipser-H, Karp-Sipser-H-scaling, Karp-Sipser-H-mindegree, and Bipartite-reduction. For $d = 3$, we also consider Local-Search [119], which replaces one hyperedge from a maximal matching M with at least two hyperedges from $E \setminus M$ to increase the cardinality of M. We did not consider local search schemes for higher dimensions or with better approximation ratios, as they are computationally too expensive. For each hypergraph, we perform ten runs of Greedy-H and Karp-Sipser-H with different random decisions and take the maximum cardinality obtained. Since Karp-Sipser-H-scaling, Karp-Sipser-H-mindegree, and Bipartite-reduction do not pick hyperedges randomly, we run them only once. We perform 20 steps of the scaling procedure in Karp-Sipser-H-scaling. We refer to quality of a matching M in a hypergraph H as the ratio of M's cardinality to the size of the smallest vertex partition of H.

On random hypergraphs

We perform experiments on two classes of d-partite, d-uniform random hypergraphs where each part has n vertices. The first class contains random

	d	k = d^{d-3}	d^{d-2}	d^{d-1}		d	k = d^{d-3}	d^{d-2}	d^{d-1}
	2	-	0.87	1.00		2	-	0.84	1.00
	3	0.80	1.00	1.00		3	0.88	1.00	1.00
$n = 10$	4	1.00	1.00	1.00	$n = 30$	4	0.99	1.00	1.00
	5	1.00	1.00	1.00		5	*	1.00	1.00
	2	-	0.88	1.00		2	-	0.87	1.00
	3	0.85	1.00	1.00		3	0.84	1.00	1.00
$n = 20$	4	1.00	1.00	1.00	$n = 50$	4	*	1.00	1.00
	5	1.00	1.00	1.00		5	*	*	*

Table 6.1: The average maximum matching cardinalities of five random instances over n on random k-out, d-partite, d-uniform hypergraphs for different k, d, and n. No runs for $k = d^{d-3}$ for $d = 2$, and the problems marked with $*$ were not solved within 24 hours.

k-out hypergraphs, and the second one contains sparse random hypergraphs.

Random k-out, d-partite, d-uniform hypergraphs

Here, we consider random k-out, d-partite, d-uniform hypergraphs described in Section 6.1. Hence (ignoring the duplicate ones), these hypergraphs have around $d \times k \times n$ hyperedges. These k-out, d-partite, d-uniform hypergraphs have been recently analyzed in the matching context by Devlin and Kahn [67]. They state in passing that k should be exponential in d for a perfect matching to exist with high probability. The bipartite graph variant of the same problem, i.e., with $d = 2$, has been extensively studied in the literature [90, 124, 125, 211].

We first investigate the existence of perfect matchings in random k-out, d-partite, d-uniform hypergraphs. For this purpose, we used CPLEX [1] to solve the hypergraph matching problem exactly and found the maximum cardinality of a matching in k-out hypergraphs with $k \in \{d^{d-3}, d^{d-2}, d^{d-1}\}$ for $d \in \{2, \ldots, 5\}$ and $n \in \{10, 20, 30, 50\}$. For each (k, d, n) triple, we created five hypergraphs and computed their maximum cardinality matchings. For $k = d^{d-3}$, we encountered several hypergraphs with no perfect matching, especially for $d = 3$. The hypergraphs with $k = d^{d-2}$ were also lacking a perfect matching for $d = 2$. However, all the hypergraphs we created with $k = d^{d-1}$ had at least one. Based on these results, we experimentally confirm Devlin and Kahn's statement. We also conjecture that d^{d-1}-out random hypergraphs have perfect matchings almost surely. The average maximum matching cardinalities we obtained in this experiment are given in Table 6.1. In this table, we do not have results for $k = d^{d-3}$ for $d = 2$, and the cases marked with $*$ were not solved within 24 hours.

We now compare the performance of the proposed heuristics on random k-out, d-partite, d-uniform hypergraphs $d \in \{3, 6, 9\}$ and $n \in \{1000, 10000\}$. We tested with k values equal to powers of 2 for $k \leq d \log d$. The results are

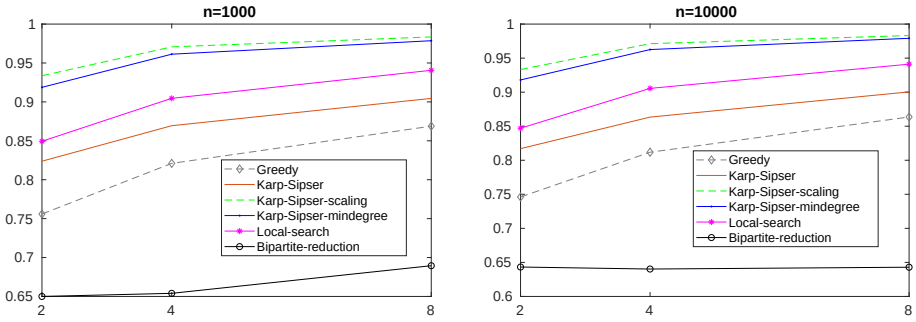
summarized in Figure 6.1. For each (k, d, n) triplet, we create ten random instances and present the average performance of the heuristics on them. The x-axis in each figure denotes k, and the y-axis reports the matching cardinality over n. As seen, Karp-Sipser-H-scaling and Karp-Sipser-H-mindegree have the best performance, comfortably beating the other alternatives. For $d = 3$ Karp-Sipser-H-scaling dominates Karp-Sipser-H-mindegree, but when $d > 3$ we see that Karp-Sipser-H-mindegree has the best performance. Local-Search performs worse than Karp-Sipser-H-scaling and Karp-Sipser-H-mindegree but better than others. Karp-Sipser-H performs better than Greedy-H. However, their performances get closer as d increases. This is due to the fact that the conditions for Rule-1 and Rule-2 hold less often for larger d, as we have more restrictions to encounter in such cases. Bipartite-reduction has worse performance than the others, and the gap in the performance grows as d increases. This happens, since at each step, we impose more and more conditions on the edges involved and there is no chance to recover from bad decisions.

Sparse random d-partite, d-uniform hypergraphs
Here, we consider a random d-partite, d-uniform hypergraph H_i created with $i \times n$ hyperedges. The parameters used for this experiment are $i \in \{1, 3, 5, 7\}$, $n \in \{4000, 8000\}$, and $d \in \{6, 9\}$ (the associated paper presents further experiments with $d = 3$) Each H_i is created by choosing the vertices of a hyperedge uniformly at random for each dimension. We do not allow duplicate hyperedges. Another random hypergraph H_{i+M} is then obtained by planting a perfect matching to H_i, that is a random perfect matching is added to the pattern of H_i. We again generate ten random instances for each parameter setting. We do not present results for Bipartite-reduction as it was always worse than the others. The average quality of different heuristics on these instances is shown in Figure 6.2. The experiments confirm that Karp-Sipser-H performs consistently better than Greedy-H. Furthermore, Karp-Sipser-H-scaling performs significantly better than Karp-Sipser-H. Karp-Sipser-H-scaling works even better than the local search heuristic [75, Fig.2]), and it is the only heuristic that is capable of finding the planted perfect matchings for a significant number of the runs. In particular, it finds a perfect matching on H_{i+M}'s in all cases except for when $d = 6$ and $i = 7$. Interestingly Karp-Sipser-H-mindegree outperforms Karp-Sipser-H-scaling on H_is but is dominated on H_{i+M}s, where it is the second best performing heuristic.

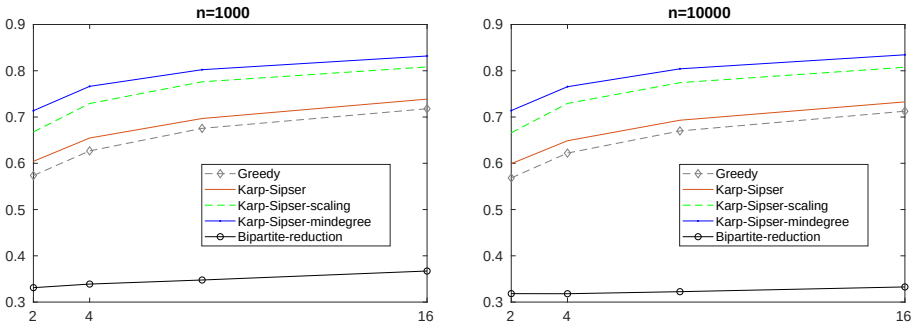
Evaluating algorithmic choices

We evaluate the use of scaling and the importance of Rule-1 and Rule-2.

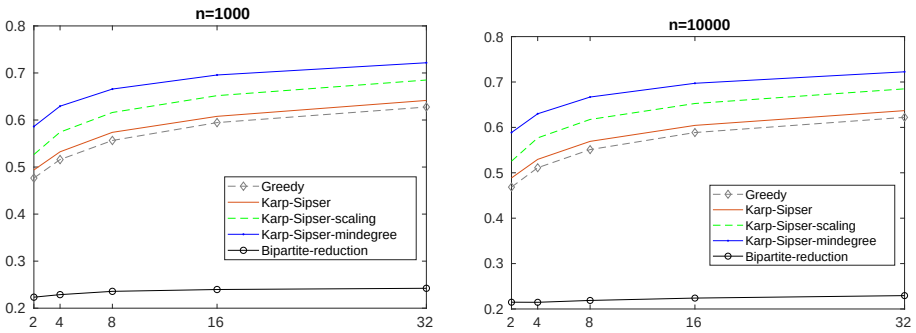
Scaling vs no-scaling
To evaluate and emphasize the contribution of scaling better, we compare the



(a) $d = 3$, $n = 1000$ (left) and $n = 10000$ (right)



(b) $d = 6$, $n = 1000$ (left) and $n = 10000$ (right)



(c) $d = 9$, $n = 1000$ (left) and $n = 10000$ (right)

Figure 6.1: The performance of the heuristics on k-out, d-partite, d-uniform hypergraphs with n vertices at each part. The y-axis is the ratio of matching cardinality to n whereas the x-axis is k. No Local-Search for $d = 6$ and $d = 9$.

	H_i: Random hypergraph								H_{i+M}: Random hypergraph + a perfect matching							
	Greedy-H		Karp-Sipser-H		Karp-Sipser-H-scaling		Karp-Sipser-H-mindegree		Greedy-H		Karp-Sipser-H		Karp-Sipser-H-scaling		Karp-Sipser-H-mindegree	
i	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000
1	0.31	0.31	0.35	0.35	0.35	0.35	0.36	0.37	0.62	0.61	0.90	0.89	1.00	1.00	1.00	1.00
3	0.43	0.43	0.47	0.47	0.48	0.48	0.54	0.54	0.51	0.50	0.56	0.55	1.00	1.00	0.99	0.99
5	0.48	0.48	0.52	0.52	0.54	0.54	0.61	0.61	0.52	0.52	0.56	0.55	1.00	1.00	0.97	0.97
7	0.52	0.52	0.55	0.55	0.59	0.59	0.66	0.66	0.54	0.54	0.57	0.57	0.84	0.80	0.71	0.70

(a) $d = 6$, without (left) and with (right) the planted matching

	H_i: Random hypergraph								H_{i+M}: Random hypergraph + a perfect matching							
	Greedy-H		Karp-Sipser-H		Karp-Sipser-H-scaling		Karp-Sipser-H-mindegree		Greedy-H		Karp-Sipser-H		Karp-Sipser-H-scaling		Karp-Sipser-H-mindegree	
i	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000
1	0.25	0.24	0.27	0.27	0.27	0.27	0.30	0.30	0.56	0.55	0.80	0.79	1.00	1.00	1.00	1.00
3	0.34	0.33	0.36	0.36	0.36	0.36	0.43	0.43	0.40	0.40	0.44	0.44	1.00	1.00	0.99	1.00
5	0.38	0.37	0.40	0.40	0.41	0.41	0.48	0.48	0.41	0.40	0.43	0.43	1.00	1.00	0.99	0.99
7	0.40	0.40	0.42	0.42	0.44	0.44	0.51	0.51	0.42	0.42	0.44	0.44	1.00	1.00	0.97	0.96

(b) $d = 9$, without (left) and with (right) the planted matching

Figure 6.2: Performance comparisons on d-partite, d-uniform hypergraphs with $n = \{4000, 8000\}$. H_i contains $i \times n$ random hyperedges, and H_{i+M} contains an additional perfect matching.

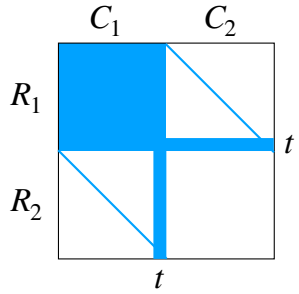


Figure 6.3: $\mathbf{A}_{KS}$: A challenging bipartite graph instance for Karp-Sipser.

t	Greedy-H	Local Search	Karp-Sipser-H	Karp-Sipser-H-scaling	Karp-Sipser-H-mindegree
2	0.53	0.99	0.53	1.00	1.00
4	0.53	0.99	0.53	1.00	1.00
8	0.54	0.99	0.55	1.00	1.00
16	0.55	0.99	0.56	1.00	1.00
32	0.59	0.99	0.59	1.00	1.00

Table 6.2: Performance of the proposed heuristics on 3-partite, 3-uniform hypergraphs corresponding to $\mathcal{X}_{KS}$ with $n = 300$ vertices in each part.

performance of the heuristics on a particular family of d-partite, d-uniform hypergraphs where their bipartite counterparts have been used before as challenging instances for the original Karp-Sipser heuristic [76].

Let $\mathbf{A}_{KS}$ be an $n \times n$ matrix discussed before in Figure 4.3 as a challenging instance to Karp-Sipser and shown in Figure 6.3 for convenience. To adapt this scheme to hypergraphs/tensors, we generate a 3-dimensional tensor $\mathcal{X}_{KS}$ such that the nonzero pattern of each marginal of the 3rd dimension is identical to that of $\mathbf{A}_{KS}$. Table 6.2 shows the performance of the heuristics (i.e., matching cardinality normalized with n) for 3-dimensional tensors with $n = 300$ and $t \in \{2, 4, 8, 16, 32\}$.

The use of scaling indeed reduces the influence of the misleading hyperedges in the dense block $R_1 \times C_1$, and the proposed Karp-Sipser-H-scaling heuristic always finds the perfect matching as does Karp-Sipser-H-mindegree. However, Greedy-H and Karp-Sipser-H perform significantly worse. Furthermore, Local-Search returns a 0.99-approximation in every case because it ends up in a local optima.

Rule-1 vs Rule-2

We finish the discussion on the synthetic data by focusing on Karp-Sipser-H. In the bipartite case, recent work [10] shows that both rules are needed to obtain perfect matchings in a class of graphs. We present a family of hypergraphs for which applying Rule-2 leads to much improved performance than applying Rule-1 only.

We use Karp-Sipser-H-R_1 to refer to Karp-Sipser-H without Rule-2. As before, we describe first the bipartite case. Let $\mathbf{A}_{RF}$ be a $n \times n$ matrix with $(\mathbf{A}_{RF})_{i,j} = 1$ for $1 \leq i \leq j \leq n$, and $(\mathbf{A}_{RF})_{2,1} = (\mathbf{A}_{RF})_{n,n-1} = 1$. That is $\mathbf{A}_{RF}$ is composed of an upper triangular matrix and two additional subdiagonal nonzeros. The first two columns and the last two rows have two nonzeros. Assume without loss of generality that the first two rows are merged by applying Rule-2 on the first column (which is discarded). Then in the reduced matrix, the first column (corresponding to the second column in the original matrix) will have one nonzero. Rule-1 can now be applied whereupon the first column in the reduced matrix will have degree one. The process continues similarly until the reduced matrix is a 2×2 dense block, where applying Rule-2 followed by Rule-1 yields a perfect matching. If only Rule-1 reductions are allowed, initially no reduction can be applied and randomly chosen edges will be matched, which negatively affects the quality of the returned matching.

For higher dimensions we proceed as follows. Let $\mathcal{X}_{RF}$ be a d-dimensional $n \times \cdots \times n$ tensor. We set $(\mathcal{X}_{RF})_{i,j,\ldots,j} = 1$ for $1 \leq i \leq j \leq n$ and $(\mathcal{X}_{RF})_{1,2,\ldots,2} = (\mathcal{X}_{RF})_{n,n-1,\ldots,n-1} = 1$. By similar reasoning, we see that Karp-Sipser-H with both reduction rules will obtain a perfect matching, whereas Karp-Sipser-H-R_1 will struggle. We give some results in Table 6.3 that show the difference between the two. We test for $n \in \{1000, 2000, 4000\}$

	d					
	2		3		6	
n	quality	$\frac{r}{n}$	quality	$\frac{r}{n}$	quality	$\frac{r}{n}$
1000	0.83	0.45	0.85	0.47	0.80	0.31
2000	0.86	0.53	0.87	0.56	0.80	0.30
4000	0.82	0.42	0.75	0.17	0.84	0.45

Table 6.3: Quality of matching and the number r of the applications of Rule-1 over n in Karp-Sipser-H-R_1, for hypergraphs corresponding to $\mathcal{X}_{RF}$. Karp-Sipser obtains perfect matchings.

and $d \in \{2, 3, 6\}$, and show the quality of Karp-Sipser-H-R_1 and the number of times that Rule-1 is applied over n. We present the best result over 10 runs. As seen in Table 6.3, Karp-Sipser-H-R_1 obtains matchings that are about 13–25% worse than Karp-Sipser-H. Furthermore, the larger the number of Rule-1 applications is, the higher the quality is.

On real-life tensors

We also evaluate the performance of the proposed heuristics on some real-life tensors selected from the FROSTT library [203]. The descriptions of the tensors are given in Table 6.4. For `nips` and `uber`, a dimension of size 17 and 24 is dropped respectively since they restrict the size of maximum cardinality matching. As described before, a d-partite, d-uniform hypergraph is obtained from a d-dimensional tensor by keeping a vertex for each dimension index, and a hyperedge for each nonzero. Unlike the previous experiments, the parts of the hypergraphs obtained from real-life tensors in Table 6.4 do not have an equal number of vertices. In this case, although the scaling algorithm works along the same lines, its output is slightly different. Let $n_i = |V_i|$ be the cardinality at ith dimension and $n_{max} = \max_{1 \le i \le d} n_i$ be the maximum one. By slightly modifying Sinkhorn-Knopp, for each iteration of Karp-Sipser-H-scaling, we scale the tensor such that the marginals in dimension i sum up to n_{max}/n_i instead of one. The results in Table 6.4 resemble those from previous sections; Karp-Sipser-H-scaling has the best performance and is slightly superior to Karp-Sipser-H-mindegree. Greedy-H and Karp-Sipser-H are close to each other and when it is feasible, Local-Search is better than them. We also see that in these instances Bipartite-reduction exhibits a good performance: its performance is at least as good as Karp-Sipser-H-scaling for the first three instances, but about 10% worse for the last one.

Tensor	d	Dimensions	Greedy-H	Local-Search	Karp-Sipser-H	Karp-Sipser-H-mindegree	Karp-Sipser-H-scaling	Bipartite-Reduction
`Uber`	3	$183 \times 1140 \times 1717$	183	183	183	183	183	183
`nips`	3	$2,482 \times 2,862 \times 14,036$	1,847	1,991	1,839	2005	2,007	2,007
`Nell-2`	3	$12,092 \times 9,184 \times 28,818$	3,913	4,987	3,935	5,100	5,154	5,175
`Enron`	4	$6,066 \times 5,699 \times 244,268 \times 1,176$	875	-	875	988	1,001	898

Table 6.4: Four real-life tensors and the performance of the proposed heuristics on the corresponding hypergraphs. No result for Local-Search for `Enron`, as it is four dimensional. `Uber` has 1,117,629 nonzeros, `nips` has 3,101,609 nonzeros, `Nell-2` has 76,879,419 nonzeros, and `Enron` has 54,202,099 nonzeros.

Using an independent set solver

We compare Karp-Sipser-H-scaling and Karp-Sipser-H-mindegree with the idea of reducing the maximum d-dimensional matching problem to that of finding an independent set in the line graph of the given hypergraph. We show that this transformation can obtain good results, but is restricted because line graphs can require too much space.

We use KaMIS [147] to find independent sets in graphs. KaMIS uses a plethora of reductions and a genetic algorithm in order to return high cardinality independent sets. We use the default settings of KaMIS (where execution time is limited to 600 seconds) and generate the line graphs with efficient sparse matrix–matrix multiplication routines. We run KaMIS, Greedy-H, Karp-Sipser-H-scaling, and Karp-Sipser-H-mindegree on a few hypergraphs from previous tests. The results are summarized in Table 6.5. The run time of Greedy-H was less than one second in all instances. KaMIS operates in rounds, and we give the quality and the run time of the first round and the final output. We note that KaMIS considers the time-limit only after the first round has been completed. As can be seen, while the quality of KaMIS is always good and in most cases superior to Karp-Sipser-H-scaling and Karp-Sipser-H-mindegree, it is also significantly slower (its principle is to deliver high quality results). We also observe that the pseudo scaling of Karp-Sipser-H-mindegree indeed helps to reduce the run time compared to Karp-Sipser-H-scaling.

The line graphs of the real-life instances from Table 6.4 are too large to be handled. We estimate using known techniques [50] the number of edges in these graphs to range from 1.5×10^{10} to 4.7×10^{13}; which translate to 126GB to 380TB of storage if edges are stored twice, using 4 bytes per edge.

	KaMIS					Greedy-H	Karp-Sipser-H-scaling		Karp-Sipser-H-mindegree	
	line graph	Round 1		Output						
hypergraph	gen. time	quality	time	quality	time	quality	quality	time	quality	time
8-out, $n = 1000$, $d = 3$	10	0.98	80	0.99	600	0.86	0.98	1	0.98	1
8-out, $n = 10000$, $d = 3$	112	0.98	507	0.99	600	0.86	0.98	197	0.98	1
8-out, $n = 1000$, $d = 9$	298	0.67	798	0.69	802	0.55	0.62	2	0.67	1
$n = 8000$, $d = 3$, H_3	1	0.77	16	0.81	602	0.63	0.76	5	0.77	1
$n = 8000$, $d = 3$, H_{3+M}	2	0.89	25	1.00	430	0.70	1.00	11	0.91	1

Table 6.5: Run time (in seconds) and performance comparisons between KaMIS, Greedy-H, and Karp-Sipser-H-scaling. The time required to create the line graphs should be added to KaMIS's overall time.

6.4 Summary, further notes and references

We have proposed heuristics for the maximum matching in d-partite, d-uniform hypergraphs problem by generalizing existing heuristics for the maximum cardinality matching in graphs. The experimental analysis on various hypergraphs (both random and corresponding to real-life tensors) show the effectiveness and efficiency of the proposed heuristics.

This being a recent study we have much future work. On the theoretical side, we plan to investigate the stated conjecture that d^{d-1}-out random hypergraphs have perfect matchings almost always, and analyze the theoretical guarantees of the proposed algorithms. On the practical side, we want to test the use of the proposed hypergraph matching heuristics in the coarsening phase of the multilevel hypergraph partitioning tools. For this, we need to adopt the proposed heuristics for the general hypergraph case (not necessarily d-partite and d-uniform). Another future work is the investigation of the weighted hypergraph matching problem, and the generalization and use of the proposed heuristics for this problem. Solvers for this problem are computational tools used in the multiple network alignment problem [179], where a matching among the vertices of more than two graphs is computed.

Part III

Ordering matrices and tensors

Chapter 7

Elimination trees and height-reducing orderings

In this chapter, we investigate the minimum height elimination problem described in Section 1.1. The formal problem definition is repeated below for convenience.

> **Minimum height elimination tree.** Given a directed graph, find an ordering of its vertices so that the associated elimination tree has the minimum height.

The standard elimination tree [200] has been used to expose parallelism in sparse Cholesky, LU, and QR factorizations [5, 8, 116, 160]. Roughly, a set of vertices without ancestor/descendant relations corresponds to a set of independent tasks that can be performed in parallel. Therefore, the total number of parallel steps, or the critical path length, is equal to the height of the tree on an unbounded number of processors [162, 214]. Obtaining an elimination tree with the minimum height for a given matrix is NP-complete [193]. Therefore, heuristic approaches are used. One set of heuristic approaches is to content oneself with the graph partitioning based methods. These methods reduce some other important cost metrics in sparse Cholesky factorization, such as the fill-in and the operation count, while giving a shallow depth elimination tree [103]. When the matrix is unsymmetric, the elimination tree for LU factorization [81] is useful to expose parallelism. In this respect, the height of the tree, again, corresponds to the number of parallel steps or the critical path length for certain factorization schemes. Here, we develop heuristics to reduce the height of elimination trees for unsymmetric matrices.

Consider the directed graph G_d obtained by replacing every edge of a given undirected graph G by two directed edges pointing at opposite directions. Then, the cycle-rank (or the minimum height of an elimination tree)

of G_d is closely related to the undirected graph parameter tree-depth [180, p.128], which corresponds to the minimum height of an elimination tree for a symmetric matrix. One can exploit this correspondence in the reverse sense in an attempt to reduce the height of the elimination tree while producing an ordering for the matrix. That is, one can use the standard undirected graph model corresponding to the matrix $\mathbf{A} + \mathbf{A}^T$, where the addition is pattern-wise. This approach of producing an undirected graph for a given directed graph by removing the direction of each edge is used in solvers such as MUMPS [8] and SuperLU [63]. This way, the state of the art graph partitioning based ordering methods can be used. We will compare the proposed heuristics with such methods.

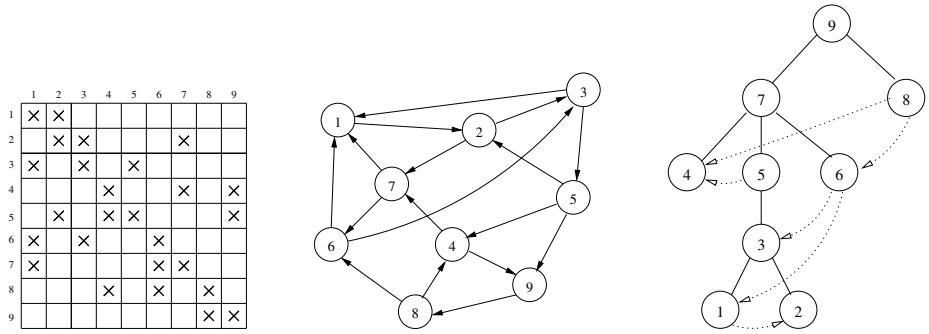
The height of the elimination tree, or the critical path length, is not the only metric for efficient parallel LU factorization of unsymmetric matrices. Depending on the factorization schemes, such as Gaussian elimination with static pivoting (implemented in GESP [157]) or multifrontal based solvers (implemented for example in WSMP [104]), other metrics come into play (such as, the fill-in, the size of blocking in GESP based solvers, e.g., SuperLU_MT [64], the maximum/minimum size of fronts for parallelism in multifrontal solvers, or the communication). Nonetheless, the elimination tree helps to give an upper bound on the performance of the suitable parallel LU factorization schemes under a fairly standard PRAM model (assuming unit size computations, infinite number of processors, and no memory/communication or scheduling overhead). Furthermore, our overall approach is based on obtaining a desirable form for LU factorization, called bordered block triangular form (or BBT form for short) [81]. This form simplifies many algorithmic details of different solvers [81, 82] and is likely to control the fill-in, as the nonzeros are constrained to be in the blocks of a BBT form. If one orders a matrix without a BBT form and then obtains this form, the fill-in can increase or decrease [83]. Hence, our BBT based approach is likely to be helpful for LU factorization schemes exploiting the BBT form as well in controlling the fill-in.

7.1 Background

We define $G^s = (V, E^s)$ as the graph associated with the directed graph G such that $E^s = \{\{v_i, v_j\} : (v_i, v_j) \in E\}$. A directed graph G is called *complete* if $(u, v) \in E$ for all vertex pairs (u, v).

Let $T = (V, r, \rho)$ be a rooted tree. An *ordering* $\sigma : V \leftrightarrow \{1, \ldots, n\}$ is a bijective function. An ordering σ is called *topological* if $\sigma(\rho(v_j)) > \sigma(v_j)$ for all $v_j \in \mathcal{V}$. Finally, a *postordering* is a special form of topological ordering in which the vertices of each subtree are ordered consecutively.

Let $\mathbf{A}$ be an $n \times n$ unsymmetric matrix with m off-diagonal nonzero entries. The standard directed graph $\mathrm{G}(\mathbf{A}) = (V, E)$ of $\mathbf{A}$ consists of a



(a) The matrix $\mathbf{A}$ (b) Directed graph G($\mathbf{A}$) (c) Elimination tree T($\mathbf{A}$)

Figure 7.1: A sample 9×9 unsymmetric matrix $\mathbf{A}$, the corresponding directed graph G($\mathbf{A}$) and elimination tree T($\mathbf{A}$).

vertex set $V = \{1, \ldots, n\}$ and an edge set $E = \{(i, j) : a_{ij} \neq 0\}$ of m edges. We adopt $G_{\mathbf{A}}$ instead of G($\mathbf{A}$) when a subgraph notion is required.

An *elimination digraph* $G_k(\mathbf{A}) = (V_k, E_k)$ is defined for $0 \leq k < n$ with the vertex set $V_k = \{k+1, \ldots, n\}$ and the edge set

$$E_k = \begin{cases} E & : k = 0 \\ E_{k-1} \cup \{(i,j) : (i,k), (k,j) \in E_{k-1}\} & : k > 0 \end{cases} .$$

We also define $\overline{V}_k = \{1, \ldots, k\}$, for each $0 \leq k < n$.

We assume that $\mathbf{A}$ is irreducible and the LU-factorization $\mathbf{A} = \mathbf{LU}$ exists where $\mathbf{L}$ and $\mathbf{U}$ are unit lower and upper triangular, respectively. Since the factors $\mathbf{L}$ and $\mathbf{U}$ are triangular, their standard directed graph models G($\mathbf{L}$) and G($\mathbf{U}$) are acyclic. Eisenstat and Liu [81] define the *elimination tree* T($\mathbf{A}$) as follows. Let

$$\textsc{Parent}(i) = \min\{j : j > i \text{ and } j \overset{\mathrm{G}(\mathbf{L})}{\Longrightarrow} i \overset{\mathrm{G}(\mathbf{U})}{\Longrightarrow} j\} ,$$

where $a \overset{\mathrm{G}}{\Rightarrow} b$ means that there is path from a to b in the graph G, then $\textsc{Parent}(i)$ corresponds to the parent of vertex i in T($\mathbf{A}$) for $i < n$, and for the root n, we put $\textsc{Parent}(n) = \infty$.

We now illustrate some of the definitions. Figure 7.1a displays a sample matrix $\mathbf{A}$ with 9 rows/columns and 26 nonzeros. Figure 7.1b shows the standard directed graph representation G($\mathbf{A}$) of this matrix. Upon elimination of vertex 1, the three edges $(3,2)$,$(6,2)$, and $(7,2)$ are added, and vertex 1 is removed, to build the elimination digraph $G_1(\mathbf{A})$. On the right side, Figure 7.1c shows the elimination tree T($\mathbf{A}$) where height $h(\mathrm{T}(\mathbf{A})) = 5$. Here, the parent vertex of 1 is 3, as $G_{\mathbf{A}}[\{1,2\}]$ is not strongly connected but $G_{\mathbf{A}}[\{1,2,3\}]$ is (thanks to the cycle $1 \to 2 \to 3 \to 1$). The cross edges of G($\mathbf{A}$) with respect to the elimination tree T($\mathbf{A}$) are represented by dashed

directed arrows in Figure 7.1c. The initial order is topological, but not postordered. However, $8, 1, 2, 3, 5, 4, 6, 7, 9$ is a postordering of $\mathrm{T}(\mathbf{A})$, and does not respect the cross edges.

7.2 The critical path length and the elimination tree height

We summarize rowwise and columnwise LU factorization schemes, also known as ikj and jik variants [70], here to be referred as *row-LU* and *column-LU*, respectively. Then, we show that the critical path lengths for the parallel row-LU and column-LU factorization schemes are equivalent to the elimination tree height whenever the matrix has a certain form.

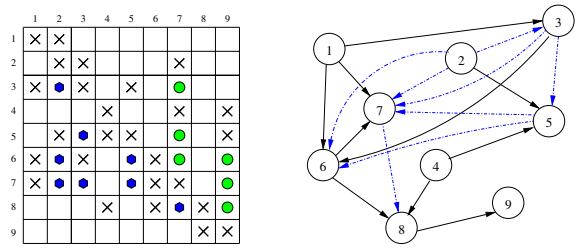
Let $\mathbf{A}$ be an $n \times n$ irreducible unsymmetric matrix, and $\mathbf{A} = \mathbf{LU}$, where $\mathbf{L}$ and $\mathbf{U}$ are unit lower and upper triangular, respectively. Algorithms 13 and 14 display the row-LU and column-LU factorization schemes, respectively. Both schemes update the given matrix $\mathbf{A}$ so that the $\mathbf{U} = [u_{ij}]_{i \leq j}$ factor is formed by the upper triangular (including the diagonal) entries of the matrix $\mathbf{A}$ at the end. The $\mathbf{L} = [\ell_{ij}]_{i \geq j}$ factor is formed by the multipliers ($\ell_{ik} = a_{ik}/a_{kk}$ for $i > k$) and a unit diagonal. We assume a coarse-grain parallelization approach [122, 160]. In Algorithm 13, task i is defined as the task of computing rows i of $\mathbf{L}$ and $\mathbf{U}$, and denoted as $\textsc{Trow}(i)$. Similarly, in Algorithm 14, task j is defined as the task of computing the columns j of both $\mathbf{L}$ and $\mathbf{U}$ factors, and denoted as $\textsc{Tcol}(j)$. The dependencies between the tasks can be represented with the directed acyclic graphs of $\mathbf{L}^{\mathrm{T}}$ and $\mathbf{U}$, for row-LU and column-LU factorization of $\mathbf{A}$, respectively.

Algorithm 13: Row-LU($\mathbf{A}$)

```
1 for i = 1 to n do
2     Trow(i):
3     for each k < i and a_ik ≠ 0 do
4         for each j > k st a_kj ≠ 0 do
5             a_ij ← a_ij − a_kj × (a_ik/a_kk)
```

Algorithm 14: Column-LU($\mathbf{A}$)

```
1 for j = 1 to n do
2     Tcol(j):
3     for each k < j st a_kj ≠ 0 do
4         for each i > k st a_ik ≠ 0 do
5             a_ij ← a_ij − a_ik × (a_kj/a_kk)
```



(a) The matrix $\mathbf{L}+\mathbf{U}$ (b) Task dependency graph $\mathrm{G}(\mathbf{L}^{\mathrm{T}})$ (c) Task dependency graph $\mathrm{G}(\mathbf{U})$

Figure 7.2: The filled matrix $\mathbf{L}+\mathbf{U}$ (a), task dependency graphs $\mathrm{G}(\mathbf{L}^{\mathrm{T}})$ for row-LU (b) and $\mathrm{G}(\mathbf{U})$ for column-LU (c).

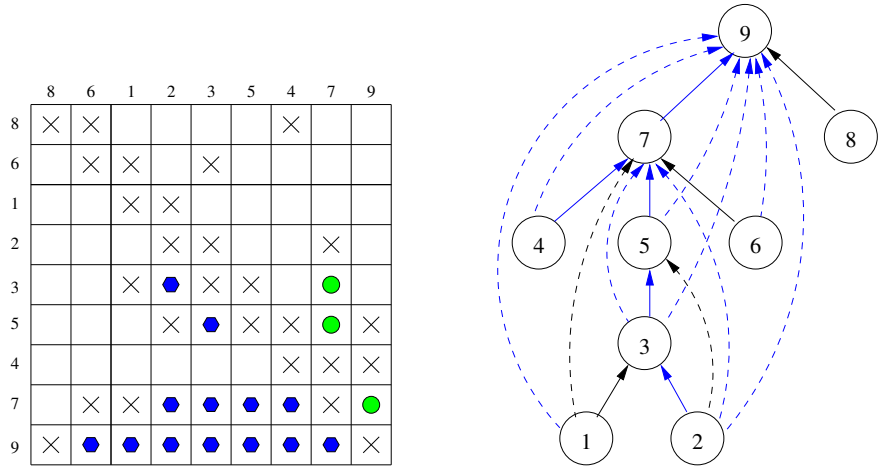
Figure 7.2 illustrates the mentioned dependencies. Figure 7.2a shows the matrix $\mathbf{L}+\mathbf{U}$ corresponding to the factors of the sample matrix $\mathbf{A}$ given in Figure 7.1a. In this figure, the blue hexagons and green circles are the fill-ins in $\mathbf{L}$ and $\mathbf{U}$, respectively. Subsequently, Figures 7.2b and 7.2c show the task dependency graphs for row-LU and column-LU factorizations, respectively. The blue dashed directed edges in Figure 7.2b, and the green dashed directed edges in Figure 7.2c correspond to the fill-ins. All edges in Figures 7.2b and 7.2c show dependencies. For example, in the column-LU, the second column depends on the first one, as the values in the first column of $\mathbf{L}$ are needed while computing the second column. This is shown in Figure 7.2c with a directed edge from the first vertex to the second one.

We now consider postorderings of the elimination tree $\mathrm{T}(\mathbf{A})$. Let $T[k]$ denote the set of vertices in the subtree of $\mathrm{T}(\mathbf{A})$ rooted at k. For a postordering, the order of siblings is not specified in general sense. A postordering is called *upper bordered block triangular (BBT)* if the topological order among the sibling vertices is respected, that is, a vertex k is numbered before a sibling vertex ℓ if there is an edge in $\mathrm{G}(\mathbf{A})$ that is directed from $T[k]$ to $T[\ell]$. Similarly, we call a postordering a *lower* BBT if the sibling vertices are ordered in reverse topological. As an example, the initial order of matrix given in Figure 7.1 is neither upper nor lower BBT, however, the order $8, 6, 1, 2, 3, 5, 4, 7, 9$ would be an upper BBT (see Figure 7.1c). The following theorem shows the relation between a BBT postordering and the task dependencies in LU factorizations.

Theorem 7.1 ([81])**.** *Assuming the edges are directed from child to parent,* $\mathrm{T}(\mathbf{A})$ *is the transitive reduction of* $\mathrm{G}(\mathbf{L}^{\mathrm{T}})$ *and* $\mathrm{G}(\mathbf{U})$ *when* $\mathbf{A}$ *is upper and lower BBT ordered, respectively.*

We follow this theorem with a corollary which provides the motivation for reducing the elimination tree height.

Corollary 7.1. *The critical path length of the task dependency graph is*



(a) The matrix $\widetilde{\mathbf{L}} + \widetilde{\mathbf{U}}$ (b) The directed graph $\mathrm{G}(\widetilde{\mathbf{L}}^{\mathrm{T}})$

Figure 7.3: The filled matrix of matrix $\widetilde{\mathbf{A}} = \widetilde{\mathbf{L}}\widetilde{\mathbf{U}}$ in BBT form (a) and the task dependency graph $\mathrm{G}(\widetilde{\mathbf{L}}^{\mathrm{T}})$ for row-LU (b).

equal to the elimination tree height $h(\mathrm{T}(\mathbf{A}))$ *for row-LU factorization when* $\mathbf{A}$ *is upper BBT ordered, and for column-LU factorization when* $\mathbf{A}$ *is lower BBT ordered.*

Figure 7.3 demonstrates the discussed points on a matrix $\widetilde{\mathbf{A}}$, which is the permuted $\mathbf{A}$ of Figure 7.1a with the upper BBT postordering $8, 6, 1, 2, 3, 5, 4, 7, 9$. Figure 7.3a and 7.3b show the filled matrix $\widetilde{\mathbf{L}} + \widetilde{\mathbf{U}}$ such that $\widetilde{\mathbf{A}} = \widetilde{\mathbf{L}}\widetilde{\mathbf{U}}$, and the corresponding task dependency graph $\mathrm{G}(\widetilde{\mathbf{L}}^{\mathrm{T}})$ for row-LU factorization, respectively. As seen in Figure 7.3b, there are only tree (solid) and back (dashed) edges with respect to elimination tree $\mathrm{T}(\widetilde{\mathbf{A}})$, due to the fact that $\mathrm{T}(\widetilde{\mathbf{A}})$ is the transitive reduction of $\mathrm{G}(\widetilde{\mathbf{L}}^{\mathrm{T}})$ (see Theorem 7.1). In the figure, the blue edges refer to fill-in nonzeros. As seen in the figure, there is a path from each vertex to the root, and a critical path $1 \to 3 \to 5 \to 7 \to 9$ has length 5 which coincides with the height of $\mathrm{T}(\widetilde{\mathbf{A}})$.

7.3 Reducing the elimination tree height

We propose a top-down approach to reorder a given matrix leading to a short elimination tree. The bigger lines of the proposed approach form a generalization of the state of the art methods used in Cholesky factorization (and are based on nested dissection [94]). We give the algorithms and discussions for the upper BBT form; they can be easily adapted for the lower BBT form. The proofs of Proposition 7.1, and Theorems 7.2–7.4 are omitted from the presentation and can be found in the original paper [136].

7.3.1 Theoretical basis

Let $\mathbf{A}$ be an $n \times n$ sparse unsymmetric irreducible matrix and $\mathrm{T}(\mathbf{A})$ be its elimination tree. In this case, a BBT decomposition of $\mathbf{A}$ can be represented as a permutation of $\mathbf{A}$ with a permutation matrix $\mathbf{P}$ such that

$$\mathbf{A}_{\mathrm{BBT}} = \begin{bmatrix} \mathrm{A}_{11} & \mathrm{A}_{12} & \dots & \mathrm{A}_{1\mathrm{K}} & \mathrm{A}_{1\mathrm{B}} \\ & \mathrm{A}_{22} & \dots & \mathrm{A}_{2\mathrm{K}} & \mathrm{A}_{2\mathrm{B}} \\ & & \ddots & \vdots & \vdots \\ & & & \mathrm{A}_{\mathrm{KK}} & \mathrm{A}_{\mathrm{KB}} \\ \mathrm{A}_{\mathrm{B1}} & \mathrm{A}_{\mathrm{B2}} & \dots & \mathrm{A}_{\mathrm{BK}} & \mathrm{A}_{\mathrm{BB}} \end{bmatrix}, \tag{7.1}$$

where $\mathbf{A}_{\mathrm{BBT}} = \mathbf{P}\mathbf{A}\mathbf{P}^T$, the number of diagonal blocks $K > 1$, and $\mathbf{A}_{\mathrm{kk}}$ is irreducible for each $1 \leq k \leq K$. The border $\mathbf{A}_{\mathrm{B}*}$ is called *minimal* if there is no permutation matrix $\mathbf{P}' \neq \mathbf{P}$ such that $\mathbf{P}'\mathbf{A}\mathbf{P}'^{\mathrm{T}}$ is a BBT decomposition and $\mathbf{A}'_{\mathrm{B}*} \subset \mathbf{A}_{\mathrm{B}*}$. That is, the border $\mathbf{A}_{\mathrm{B}*}$ is minimal if we cannot remove columns from $\mathbf{A}_{\mathrm{B}*}$ and the corresponding rows from $\mathbf{A}_{\mathrm{B}*}$, permute them together to a diagonal block, and still have a BBT form. We give the following proposition aimed to be used for Theorem 7.2.

Proposition 7.1. *Let* $\mathbf{A}$ *be in upper BBT form and the border* $\mathbf{A}_{\mathrm{B}*}$ *be minimal. The elimination digraph* $G_\kappa(\mathbf{A})$ *is complete where* $\kappa = \sum_{i=1}^{K} |\mathbf{A}_{\mathrm{kk}}|$.

The following theorem provides a basis for reducing the elimination tree height $h(\mathrm{T}(\mathbf{A}))$.

Theorem 7.2. *Let* $\mathbf{A}$ *be in upper BBT form, and the border* $\mathbf{A}_{\mathrm{B}*}$ *be minimal. Then,*

$$h(\mathrm{T}(\mathbf{A})) = |\mathbf{A}_{\mathrm{B}*}| + \max_{1 \leq k \leq K} h(\mathrm{T}(\mathbf{A}_{\mathrm{kk}})) \ .$$

The theorem says that the critical path length in the parallel LU factorization methods summarized in Section 7.2 can be expressed recursively in terms of the border and the critical path length of a block with the maximum size. This holds for the row-LU scheme when the matrix $\mathbf{A}$ is in an upper BBT form, and for the column-LU scheme, when $\mathbf{A}$ is in a lower BBT form. Hence, having a small border size and diagonal blocks of similar size is likely to lead to reduced critical path length.

7.3.2 Recursive approach: Permute

We propose a recursive approach to reorder a given matrix so that the elimination tree is reduced. Algorithm 15 gives the overview of the solution framework. The main procedure Permute takes an irreducible unsymmetric matrix $\mathbf{A}$ as its input. It calls PermuteBBT to decompose $\mathbf{A}$ into a BBT form, $\mathbf{A}_{\mathrm{BBT}}$. Upon obtaining such a decomposition, the main procedure calls itself on each diagonal block $\mathbf{A}_{\mathrm{kk}}$, recursively. Whenever the

matrix becomes sufficiently small (its size gets smaller than τ), the procedure PermuteBT is called in order to compute a fine BBT decomposition in which the diagonal blocks are of unit size, here to be referred as *bordered triangular* (BT) decomposition, and the recursion terminates.

Algorithm 15: Permute($\mathbf{A}$)

```
1 if |A| < τ then                                  /* Recursion terminates */
2 |   PermuteBT(A)
3 else
4 |   A_BBT ← PermuteBBT(A) /* As given in (7.1)                          */
5 |   for k ← 1 to K do
6 |   |   Permute(A_kk) /* Subblocks of A_BBT                             */
  |___|
```

7.3.3 BBT decomposition: PermuteBBT

Permuting $\mathbf{A}$ into a BBT form translates into finding a *strong (vertex) separator* of G($\mathbf{A}$), where the strong separator itself corresponds to the border $\mathbf{A}_{\mathrm{B}*}$. Regarding to Theorem 7.2, we are particularly interested in finding minimal borders.

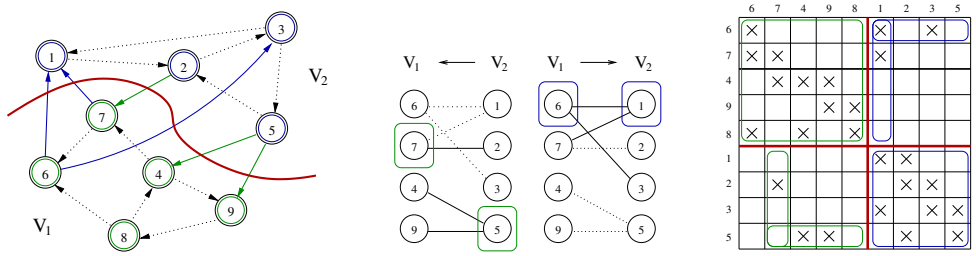
Let $G = (V, E)$ be a strongly connected directed graph. A strong separator (also called *directed vertex separator* [138]) is defined as a vertex set $S \subset V$ such that $G - S$ is not strongly connected. Moreover, S is said to be *minimal* if $G - S'$ is strongly connected for any proper subset $S' \subset S$.

The algorithm that we propose to find a strong separator is based on bipartitioning of directed graphs. In case of undirected graphs, typically, there are two kinds of graph partitioning methods which differ by their separators: *edge separators* and *vertex separators*. An edge (or vertex) separator is a set of edges (or vertices) whose removal leaves the graph disconnected.

A BBT decomposition of a matrix may have three or more subblocks (as seen in (7.1)). However, the algorithms we use to find a strong separator utilize 2-way partitioning, which in turn constructs two parts, each of which may potentially have several components. For the sake of simplicity in the presentation, we assume that both parts are strongly connected.

We introduce some notation in order to ease the discussion. For a pair of vertex subsets (U, W) of directed graph G, we write $U \mapsto W$, if there *may* be some edges from U to W, but there is *no* edge from W to U. Besides, $U \leftrightarrow W$ and $U \not\leftrightarrow W$ are used in a more natural way, that is, $U \leftrightarrow W$ implies that there *may* be edges in both directions, whereas $U \not\leftrightarrow W$ refers to absence of such an edge between U and W.

We cast our problem of finding strong separators as follows: For a given directed graph G, find a vertex partition $\Pi^* = \{V_1, V_2, S\}$, where S refers to a strong separator, and V_1, V_2 refer to vertex parts, so that S has small



(a) An edge separator $\Pi_{ES} = \{V_1, V_2\}$

(b) Covers for $E_{2\mapsto 1}$ (left) and $E_{1\mapsto 2}$ (right)

(c) Matrix view of Π_{ES}

Figure 7.4: A sample 2-way partition $\Pi_{ES} = \{V_1, V_2\}$ by edge separator (a), bipartite graphs and covers for directed cuts $E_{2\mapsto 1}$ and $E_{1\mapsto 2}$ (b), and the matrix view of Π_{ES}.

cardinality, the part sizes are close to each other, and either $V_1 \mapsto V_2$ or $V_2 \mapsto V_1$.

7.3.4 Edge-separator-based method

The first step of the method is to obtain a 2-way partition $\Pi_{ES} = \{V_1, V_2\}$ of G by edge separator. We build two strong separators upon Π_{ES} and choose the one with the smaller cardinality.

For an edge separator $\Pi_{ES} = \{V_1, V_2\}$, the *directed cut* $E_{i\mapsto j}$ is defined as the set of edges directed from V_i to V_j, i.e., $E_{i\mapsto j} = \{(u, v) \in E : u \in V_i,\ v \in V_j\}$, and we say that a vertex set S *covers* $E_{i\mapsto j}$ if for any edge $(u, v) \in E_{i\mapsto j}$, either $u \in S$ or $v \in S$, for $i \neq j \in \{1, 2\}$.

Initially, we have $V_1 \leftrightarrow V_2$. Our goal is to find a subset of vertices $S \subset V$ so that either $\widetilde{V}_1 \mapsto \widetilde{V}_2$ or $\widetilde{V}_2 \mapsto \widetilde{V}_1$, where $\widetilde{V}_1$ and $\widetilde{V}_2$ are the sets of remaining vertices in V_1 and V_2 after the removal of those vertices in S, that is, $\widetilde{V}_1 = V_1 - S$ and $\widetilde{V}_2 = V_2 - S$. The following theorem serves to the purpose of finding such a subset S.

Theorem 7.3. *Let $G = (V, E)$ be a directed graph, $\Pi_{ES} = \{V_1, V_2\}$ be an edge separator of G and $S \subseteq V$. Now, $\widetilde{V}_i \mapsto \widetilde{V}_j$ if and only if S covers $E_{j\mapsto i}$, for $i \neq j \in \{1, 2\}$.*

The problem of finding a strong separator S can be encoded as finding a minimum vertex cover in the bipartite graph whose edge set is populated only by the edges of $E_{j\mapsto i}$. Algorithm 16 gives the pseudocode. As seen in the algorithm, we find two separators $S_{1\mapsto 2}$ and $S_{2\mapsto 1}$, which cover $E_{2\mapsto 1}$ and $E_{1\mapsto 2}$, and result in $\widetilde{V}_1 \mapsto \widetilde{V}_2$ and $\widetilde{V}_2 \mapsto \widetilde{V}_1$, respectively. At the end, we take the strong separator with the smaller cardinality.

In this method, how the edge separator is found matters. The objective for Π_{ES} is to minimize the cutsize $\Psi(\Pi_{ES})$ which is typically defined over

Algorithm 16: FindStrongVertexSeparatorES(G)

```
1 Π_ES ← GraphBipartiteES(G) /* Π_ES = {V_1, V_2}                    */
2 Π_{1↦2} ← FindMinCover(G, E_{2↦1})
3 Π_{2↦1} ← FindMinCover(G, E_{1↦2})
4 if |S_{1↦2}| < |S_{2↦1}| then
5 |   return Π_{1↦2} = {Ṽ_1, Ṽ_2, S_{1↦2}} /* Ṽ_1 ↦ Ṽ_2            */
6 else
7 |   return Π_{2↦1} = {Ṽ_1, Ṽ_2, S_{2↦1}} /* Ṽ_2 ↦ Ṽ_1            */
```

the cut edges as follows:

$$\Psi_{es}(\Pi_{ES}) = |E_c| = |\{(u,v) \in E : \pi(u) \neq \pi(v)\}| \,, \tag{7.2}$$

where $u \in V_{\pi(u)}$ and $v \in V_{\pi(v)}$, and E_c refers to set of cut edges. We note that the use of the above metric in Algorithm 16 is a generalization of a method used for symmetric matrices [161]. However, $\Psi_{es}(\Pi_{ES})$ is loosely related to the size of the cover found in order to build the strong separator. A more suitable metric would be the number of vertices from which a cut edge emanates or the number of vertices to which a cut edge is directed. Such a cutsize metric can be formalized as follows:

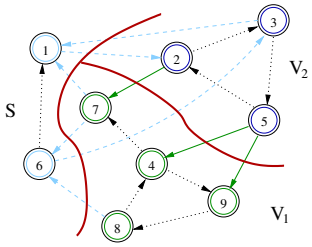
$$\Psi_{cn}(\Pi_{ES}) = |V_c| = |\{v \in V : \exists (u,v) \in E_c\}| \,. \tag{7.3}$$

The following theorem shows the close relation between $\Psi_{cn}(\Pi_{ES})$ and the size of the strong separator S.

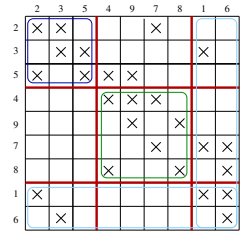
Theorem 7.4. *Let $G = (V, E)$ be a directed graph, $\Pi_{ES} = \{V_1, V_2\}$ be an edge separator of G, and $\Pi^* = \{\widetilde{V}_1, \widetilde{V}_2, S\}$ be the strong separator built upon Π_{ES}. Then, $|S| \leq \Psi_{cn}(\Pi_{ES})/2$.*

Apart from the theorem, we also note that optimizing (7.3) is likely to yield bipartite graphs which are easier to cover than those resulting from optimizing (7.2). This is because of the fact that all edges emanating from a single vertex or many different vertices are considered as the same with (7.2), whereas those emanating from a single vertex are preferred in (7.3).

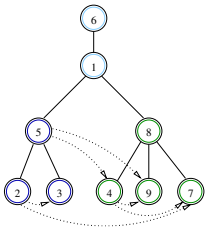
We note that the cutsize metric as given in (7.3) can be modeled using hypergraphs. The hypergraph that models this cutsize metric (called the column-net hypergraph model [39]) has the same vertex set as the directed graph $G = (V, E)$, and a hyperedge for each vertex $v \in V$ that connects all $u \in V$ such that $(u, v) \in E$. Any partition of the vertices of this hypergraph that minimizes the number of hyperedges in the cut induces a vertex partition on G, with an edge separator that minimizes the cutsize according to the metric (7.3). A similar statement holds for another hypergraph constructed by reversing the edge directions (called the row-net model [39]).



(a) Strong separator $\Pi_{2\mapsto 1}$



(b) $\mathbf{A}_{\mathrm{BBT}}$, matrix view of $\Pi_{2\mapsto 1}$



(c) Elimination tree $\mathrm{T}(\mathbf{A}_{\mathrm{BBT}})$

Figure 7.5: $\Pi_{2\mapsto 1}$, the strong separator built upon $S_{2\mapsto 1}$, the cover of $E_{1\mapsto 2}$ given in 7.4b (a), the matrix $\mathbf{A}_{\mathrm{BBT}}$ obtained by $\Pi_{2\mapsto 1}$ (b) and the elimination tree $\mathrm{T}(\mathbf{A}_{\mathrm{BBT}})$.

Figures 7.4 and 7.5 give an example for finding a strong separator based on an edge separator. In Figure 7.4a, the red curve implies the edge separator $\Pi_{ES} = \{V_1, V_2\}$ such that $V_1 = \{4, 6, 7, 8, 9\}$, the green vertices, and $V_2 = \{1, 2, 3, 5\}$, the blue vertices. The cut edges of $E_{2\mapsto 1} = \{(2, 7), (5, 4), (5, 9)\}$ and $E_{1\mapsto 2} = \{(6, 1), (6, 3), (7, 1)\}$ are given in color blue and green, respectively. We see two covers for the bipartite graphs corresponding to each of the directed cuts $E_{2\mapsto 1}$ and $E_{1\mapsto 2}$ in Figure 7.4b. As seen in Figure 7.4c, these directed cuts refer to nonzeros of the off-diagonal blocks in matrix view. Figure 7.5a gives the strong separator $\Pi_{2\to 1}$ built upon Π_{ES} by the cover $S_{2\to 1} = \{1, 6\}$ of the bipartite graph corresponding to $E_{1\to 2}$, which is given on the right of 7.4b. Figures 7.5b and 7.5c depict the matrix view of the strong separator $\Pi_{2\to 1}$, and the corresponding elimination tree, respectively. It is worth mentioning that in matrix view, V_2 proceeds V_1, since we use $\Pi_{2\to 1}$ instead of $\Pi_{1\to 2}$. As seen in 7.5c, since the permuted matrix (in Figure 7.5b) has an upper BBT postordering, all cross edges of the elimination tree are headed from left to right.

7.3.5 Vertex-separator-based method

A vertex separator is a strong separator restricted so that $V_1 \not\leftrightarrow V_2$. This method is based on refining the separator so that either $V_1 \mapsto V_2$ or $V_1 \mapsto V_2$. The vertex separator can be viewed as a 3-way vertex partition $\Pi_{VS} = \{V_1, V_2, \widehat{S}\}$. As in the previous method based on edge separators, we build two strong separators upon Π_{VS} and select the one having the smaller strong separator.

The criterion used to find an initial vertex separator can be formalized as

$$\Psi_{vs}(\Pi_{VS}) = |\widehat{S}| \ . \tag{7.4}$$

Algorithm 17 gives the overall process to obtain a strong separator. This

algorithm follows Algorithm 16 closely. Here, the procedure that refines the initial vertex separator, namely RefineSeparator, works as follows. It visits the separator vertices once, and moves the vertex at hand, if possible, to V_1 or V_2 so that $V_i \mapsto V_j$ after the movement, where $i \mapsto j$ is specified as either $1 \mapsto 2$ or $2 \mapsto 1$.

Algorithm 17: FindStrongVertexSeparatorVS(G)

```
1 Π_VS ← GraphBipartiteVS(G) /* Π_VS = {V_1, V_2, Ŝ}                    */
2 Π_{1→2} ← RefineSeparator(G, Π_VS, 1 → 2)
3 Π_{2→1} ← RefineSeparator(G, Π_VS, 2 → 1)
4 if |S_{1→2}| < |S_{2→1}| then
5 |   return Π_{1→2} = {Ṽ_1, Ṽ_2, S_{1→2}} /* Ṽ_1 → Ṽ_2               */
6 else
7 |   return Π_{2→1} = {Ṽ_1, Ṽ_2, S_{2→1}} /* Ṽ_2 → Ṽ_1               */
```

7.3.6 BT decomposition: PermuteBT

The problem of permuting $\mathbf{A}$ into a BT form can be decoded as finding a *feedback vertex set*, which is defined as a subset of vertices whose removal makes a directed graph acyclic. In matrix view, a feedback vertex set corresponds to border $\mathbf{A}_{\mathrm{B}*}$ when the matrix is permuted into a BBT form (7.1), where each $\mathbf{A}_{ii}$, for $i = 1, \ldots, K$, is of unit size.

The problem of finding a feedback vertex set of size less than a given value is NP-complete [126], but fixed-parameter tractable [44]. In literature, there are greedy algorithms that perform well in practice [152, 184]. In this work, we adopt the algorithm proposed by Levy and Low [152].

For example, in Figure 7.5, we observe that the sets $\{5\}$ and $\{8\}$ are used as the feedback-vertex sets for $\widetilde{V}_1$ and $\widetilde{V}_2$, respectively, which is the reason those rows/columns are permuted last in their corresponding subblocks.

7.4 Experiments

We investigate the performance of the proposed heuristic, here to be referred as `BBT`, in terms of the elimination tree height and ordering time. We have three variants of `BBT`: (i) `BBT-es`, based on minimizing the edge cut (7.2); (ii) `BBT-cn`, based on minimizing the hyperedge cut (7.3); (iii) `BBT-vs`, based on minimizing the vertex separator (7.4). As a baseline, we used `MeTiS` [130] on the symmetrized matrix $\mathbf{A} + \mathbf{A}^T$. `MeTiS` uses state of the art methods to order a symmetric matrix and leads to short elimination trees. Its use in the unsymmetric case on $\mathbf{A} + \mathbf{A}^T$ is a common practice. `BBT` is implemented in `C` and compiled with `mex` of MATLAB which uses `gcc` 4.4.5. We used MeTiS library for graph partitioning according to the cutsize metrics (7.2)

and (7.4). For the edge cut metric (7.2), we input MeTiS the graph of $\mathbf{A} + \mathbf{A}^T$, where the weight of the edge $\{i, j\}$ is 2 (both $a_{ij}, a_{ji} \neq 0$) or 1 (either $a_{ij} \neq 0$ or $a_{ji} \neq 0$, but not both). For the vertex separator metric (7.4), we input the graph of $\mathbf{A}+\mathbf{A}^T$ (no weights) to the corresponding MeTiS procedure. We implemented the cutsize metric given in (7.3) using PaToH [40] (in the `default` setting) for hypergraph partitioning using the column-net model (we did not investigate the use of row-net model). We performed the experiments on an AMD Opteron Processor 8356 with 32 GB of RAM.

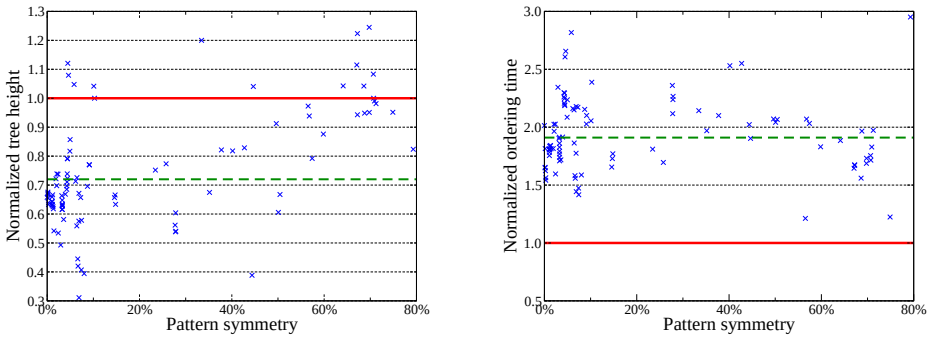
In the experiments, for each matrix we detected dense rows and columns, where a row/column is deemed to be dense if it has at least $10\sqrt{n}$ nonzeros. We applied MeTiS and `BBT` on the submatrix of non-dense rows/columns, and produced the final ordering by appending the dense rows/columns after the permutations obtained for the submatrices. The performance results are computed as the median of eleven runs.

Recall from Section 7.3.2 that whenever the size of the input matrix is smaller than the parameter τ, the recursion terminates in `BBT` variants and a feedback-vertex-set-based BT decomposition is applied. We first investigate the effect of τ in order to suggest a concrete approach. For this purpose, we build a dataset of matrices, called `UFL` below. The matrices in `UFL` are from the SuiteSparse Matrix Collection [59] and chosen with the following properties: square, 5K $\leq n \leq$ 100K, $nnz \leq$ 1M, real, and not of type `Graph`. These criteria enable an automatic selection of matrices from different application domains without having to specify the matrices individually. Other criteria could be used; ours help to select a large set of matrices each of which is not too small and not too large to be handled easily within Matlab. We exclude matrices recorded as `Graph` in the SuiteSparse collection, because most of these matrices have nonzeros from a small set of integers (for example $\{-1, 1\}$) and are reducible. We further detect matrices with the same nonzero pattern in the `UFL` dataset and keep only one matrix per unique nonzero pattern. We then preprocess the matrices by applying `MC64` [72] in order to permute the columns so that the permuted matrix has a maximum diagonal product. Then, we identify the irreducible blocks using Dulmage-Mendelsohn decomposition [77], permute the matrices to block diagonal form and delete all entries that lie in the off-diagonal blocks. This type of preprocessing is common in similar studies [83], and corresponds to preprocessing for numerics and reducibility. After this preprocessing, we discard the matrices whose total (remaining) number of nonzeros is less than twice its size, or whose pattern symmetry is greater than 90%, where the pattern symmetry score is defined as $(nnz(\mathbf{A})-n)/(nnz(\mathbf{A}+\mathbf{A}^{\mathrm{T}})-n)-0.5$. We ended up with 99 matrices in the `UFL` dataset.

In Table 7.1, we summarize the performance of the `BBT` variants as normalized to that of MeTiS on the `UFL` dataset for the recursion termination parameter $\tau \in \{3, 50, 250\}$. The table presents the geometric means of per-

	Tree Height ($h(\mathbf{A})$)			Ordering Time		
τ	`-es`	`-cn`	`-vs`	`-es`	`-cn`	`-vs`
3	0.81	0.69	0.68	1.68	3.53	2.58
50	0.84	0.71	0.72	1.26	2.71	1.91
250	0.93	0.82	0.86	1.12	2.12	1.43

Table 7.1: Statistical indicators of the performance of `BBT` variants on the `UFL` dataset normalized to that of `MeTiS` with recursion termination parameter $\tau \in \{3, 50, 250\}$.



(a) `BBT-vs` / `MeTiS` Tree Height ($h(\mathbf{A})$) (b) `BBT-vs` / `MeTiS` Ordering Time

Figure 7.6: `BBT-vs` / `MeTiS` performance on the `UFL` dataset. Dashed, green lines mark the geometric mean of the ratios.

formance results over all matrices of the dataset. As seen in the table, for smaller values of τ, all `BBT` variants achieve shorter elimination trees at a cost of increased processing time to order the matrix. This is because of the fast but probably not of very high quality local ordering heuristic—as discussed before, the heuristic here does not directly consider the height of the subtree corresponding to the submatrix as an optimization goal. Other τ values could also be tried but we find $\tau = 50$ as a suitable choice in general, as it performs close to $\tau = 3$ in quality and close to $\tau = 250$ in efficiency.

From Table 7.1, we observe that `BBT-vs` method performs in the middle of the other two `BBT` variants, but close to `BBT-cn`, in terms of the tree height. All the three variants improve the height of the tree with respect to `MeTiS`. Specifically, at $\tau = 50$, `BBT-es`, `BBT-cn`, and `BBT-vs` obtain improvements of 16%, 29% and 28%, respectively. The graph based methods `BBT-es` and `BBT-vs` run slower than `MeTiS`, as they contain additional overheads of obtaining the covers and refining the separators of $\mathbf{A} + \mathbf{A}^T$ for $\mathbf{A}$. For example, at $\tau = 50$, the overheads result in 26% and 171% longer ordering time for `BBT-es` and `BBT-vs`, respectively. As expected, the hypergraph based method `BBT-cn` is much slower than others (for example, 112% longer ordering time with respect to `MeTiS` with $\tau = 50$). Since we

identify BBT-vs as fast and of high quality (it is close to the best BBT variant in terms of the tree height and the fastest one in terms of the run time), we display its individual performance results in detail, with $\tau = 50$, in Figure 7.6. In Figures 7.6a and 7.6b, each individual point represents the ratio BBT-vs / MeTiS, in terms of tree height and ordering time, respectively. As seen in Figure 7.6a, BBT-vs reduces the elimination tree height on 85 out of 99 matrices in the UFL dataset, and achieves a reduction of 28%, in terms of the geometric mean, with respect to MeTiS (the green dashed line in the figure represents the geometric mean). We observe that the pattern symmetry and the reduction in the elimination tree height highly correlate after a certain pattern symmetry score. More precisely, the correlation coefficient between the pattern symmetry and the normalized tree height is 0.6187 for those matrices with pattern symmetry greater than 20%. This is of course in agreement with the fact that LU factorization methods exploiting the unsymmetric pattern have much to gain on matrices with lower pattern symmetry score and less to gain otherwise than the alternatives. On the other hand, as seen in Figure 7.6b, BBT-vs performs consistently slower than MeTiS, with an average of 91% slow down.

The fill-in generally increases [83] when a matrix is reordered into a BBT form with respect to the original ordering using the elimination tree. We noticed that BBT-vs almost always increases the number of nonzeros in $\mathbf{L}$ (by 157% with respect to MeTiS in a set of matrices). However, the number of nonzeros in $\mathbf{U}$ almost always decreases (by 15% with respect to MeTiS in the same dataset). This observation may be exploited during Gaussian elimination where $\mathbf{L}$ is not stored.

7.5 Summary, further notes and references

We investigated the elimination tree model for unsymmetric matrices as a means of capturing dependencies among the tasks in row-LU and column-LU factorization schemes. Specifically, we focused on permuting a given unsymmetric matrix to obtain an elimination tree with reduced height in order to expose higher degree of parallelism by minimizing the critical path length in parallel LU factorization schemes. Based on the theoretical findings, we proposed a heuristic, which orders a given matrix to a bordered block diagonal form with a small border size and blocks of similar sizes, and then locally orders each block. We presented three variants of the proposed heuristic. These three variants achieved, on average, 24%, 31%, and 35% improvements with respect to a common method of using MeTiS (a state of the art tool used for the Cholesky factorization) on a small set of matrices. On a larger set of matrices, the performance improvements were 16%, 28%, and 29%. The best performing variant is about 2.71 times slower than MeTiS on the larger set, while others are slower by factors of 1.26 and 1.91.

Although numerical methods implementing the LU factorization with the help of the standard elimination tree are well developed, those implementing the LU factorization using the elimination discussed in this section are lacking. A potential implementation will have to handle many scheduling issues and therefore the use of a runtime system seems adequate. On the combinatorial side, we believe that there is a need for a direct method to find strong separators, which would give better performance in terms of the quality and the run time.

Chapter 8

Sparse tensor ordering

In this chapter, we investigate the tensor ordering problem described in Section 1.5. The formal problem definition is repeated below for convenience.

> **Tensor ordering.** Given a sparse tensor $\mathcal{X}$, find an ordering of the indices in each dimension so that the nonzeros of $\mathcal{X}$ have indices close to each other.

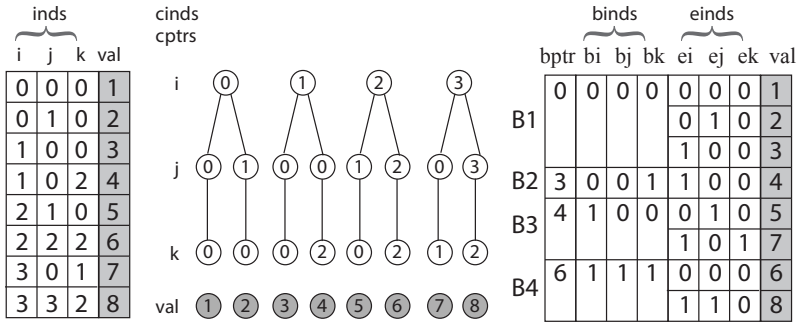
We propose two ordering algorithms for this problem. The first proposed heuristic BFS-MCS is a breadth first search (BFS)-like approach based on the maximum cardinality search family and works on the hypergraph representation of the given tensor. The second proposed heuristic Lexi-Order is an extension of doubly lexical ordering of matrices to tensors. We show the effects of these schemes in the MTTKRP performance when the tensors are stored in three existing sparse tensor formats: coordinate (COO), compressed sparse fiber (CSF), and hierarchical coordinate (HiCOO). The associated paper [156] has further contributions with respect to parallelization and includes results on parallel implementations of MTTKRP with the three formats, as well as runs with Candecomp/Parafac decomposition algorithms.

8.1 Three sparse tensor storage formats

We consider the three state-of-the-art tensor formats (COO, CSF, HiCOO) which are all for general unstructured sparse tensors.

Coordinate format (COO)

The coordinate (COO) format is the simplest yet arguably most popular format by far. It stores each nonzero value along with all of its position indices, shown in Figure 8.1a. i, j, k are indices (inds) of the nonzeros stored in the val array.



(a) COO (b) CSF (c) HiCOO

Figure 8.1: Three common formats for sparse tensors (example is originally from [155]).

Compressed sparse fibers (CSF)

Compressed Sparse Fiber (CSF) is a hierarchical, fiber-centric format that effectively generalizes the CSR matrix format to tensors. An example of its representation appears in Figure 8.1b. Conceptually, CSF organizes nonzero indices into a tree. Each level corresponds to a tensor mode, and each nonzero is a path from the root to a leaf. The indices of CSF are stored in cinds with the pointers stored in cptrs to indicate the locations of nonzeros at the next level. Since cptrs needs to show the range up to nnz, we use β_{int} or β_{long} accordingly for differently sized-tensors.

Hierarchical coordinate format (HiCOO)

Hierarchical Coordinate (HiCOO) [155] format derives from COO format, but improves upon it by compressing the indices in units of sparse tensor blocks. HiCOO stores a sparse tensor in a sparse-blocked pattern with a pre-specified block size B, meaning in $B \times \cdots \times B$ blocks. It represents every block by compactly storing its nonzero triples using fewer bits. Figure 8.1c shows the example tensor given $2 \times 2 \times 2$ blocks ($B = 2$). For a third-order tensor, bi, bj, bk are *block indices* indexing tensor blocks; ei, ej, ek are *element indices* indexing nonzeros within a tensor block. A bptr array stores the pointers of each block's beginning locations, and val saves all the nonzero values. The block ratio α_b of a HiCOO representation of tensor is defined as the number of blocks divided by the number of nonzeros in the tensor. The average slice size per tensor block $\overline{c_b}$ of a HiCOO representation of tensor is defined as the average slice size per tensor block. These two key parameters describe the effectiveness of storing a tensor in the HiCOO format.

Comparing the three tensor formats, HiCOO and COO treat every mode equally and do not assume any mode order, these preserve the mode-generic

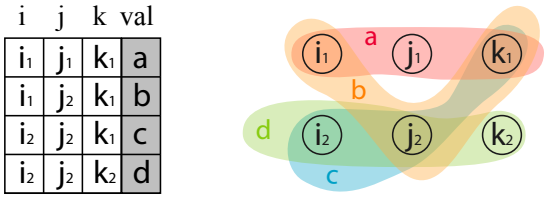


Figure 8.2: A sparse tensor and the associated hypergraph.

orientation [155]. CSF has a strong mode-specificity, since the tree structure implies a mode ordering for enumeration of nonzeros. Besides, comparing to COO format, HiCOO and CSF save storage space and memory footprints whereas achieve higher performance generally.

8.2 Problem definition and solutions

Our objective is to improve the performance of Mttkrp and therefore CPD algorithms based on the three tensor storage formats described above. We will achieve this objective by ordering (or relabeling) the indices in one or more modes of the input tensor so as to improve the data locality in the tensor and the factor matrices of an Mttkrp operation. Take for example two nonzeros (i_2, j_2, k_1) and (i_2, j_2, k_2) of a third-order tensor in Figure 8.2. Relabeling k_1 and k_2 to other two indices close to each other will potentially improve cache hits for both tensor and their corresponding rows of factor matrices in the Mttkrp operation. Besides, it will also influence the tensor storage for some formats. (Analysis will be illustrated in Section 8.2.3.)

We propose two ordering heuristics. The aim of the heuristics is to arrange the nonzeros close to each other, in all modes. If we were to look at matrices, this would correspond to reordering the rows and columns so that all nonzeros are clustered around the diagonal. This way, nonzeros in a row or column would be close to each other, and any blocking (by imposing fixed sized blocks) would have nonzero blocks only around the diagonal. The proposed heuristics are based on these observations and try to obtain similar behavior for tensors. The output of a reordering algorithm is the permutations for all modes being used to relabel tensor indices of them.

8.2.1 BFS-MCS

BFS-MCS is a breadth first search (BFS)-like heuristic approach based on the maximum cardinality search family [207]. We first construct the d-partite, d-uniform hypergraph for a sparse tensor as described in Section 1.5. Recall that in this hypergraph, the vertices are tensor indices in all modes and hyperedges represent its nonzero entries. Figure 8.2 shows the hyper-

graph associated with a sparse tensor. Here, the vertices are blank circles, and hyperedges are represented by grouping vertices.

For an Nth-order sparse tensor, we need to find the permutations for N modes. We determine a permutation for a given mode n (perm_n) of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ as follows. Suppose some of the indices of mode n are already ordered. Then, BFS-MCS picks the next to-be-ordered index as the one with the strongest connection to the currently ordered index set. In the case of ties, it selects the index with the smallest number of nonzeros in the corresponding sub-tensor. Intuitively, a stronger connection represents more common indices in the modes other than n among an unordered vertex and the already ordered ones. This means more data from factor matrices can be reused in MTTKRP, if found in cache.

Algorithm 18: BFS-MCS ordering based on maximum cardinality search for a given mode.

Data: An Nth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, hypergraph $G = (V, E)$, mode n
Result: Permutation $perm_n$

1 $\mathbf{w}_p(i) \leftarrow 0$ for $i = 1, \ldots, I_n$
2 $\mathbf{w}_s(i) \leftarrow -|\mathcal{X}(\ldots, i, \ldots)|$ `/* number of nonzeros with` i `in mode` n `*/`
3 Build max-heap H_n for mode-n indices with $\mathbf{w}_p$ and $\mathbf{w}_s$ as the primary and secondary keys respectively.
4 $\mathbf{m}_V(i) \leftarrow 0$ for $i = 1, \ldots, I_n$
5 **for** $i_n = 1, \ldots, I_n$ **do**
6 $v_n^{(0)} \leftarrow \text{GetHeapMax}(H_n)$
7 $perm_n(v_n^{(0)}) \leftarrow i_n$
8 $\mathbf{m}_V(v_n^{(0)}) \leftarrow 1$
9 **for** $e_n \in$ *hyperedges of* $v_n^{(0)}$ **do**
10 **for** $v \in e_n$, v *is not in mode* n *and* $\mathbf{m}_V(v) = 0$ **do**
11 $\mathbf{m}_V(v) \leftarrow 1$
12 **for** $e \in$ *hyperedges of* v *and* $e \neq e_n$ **do**
13 $v_n \leftarrow$ vertex in mode n of e
14 **if** inHeap(v_n) **then**
15 $\mathbf{w}_p(v_n) \leftarrow \mathbf{w}_p(v_n) + 1$
16 heapUpdateKey($H_n, \mathbf{w}_p(v_n)$)
17 **return** $perm_n$;

The process is implemented for all mode-n indices by maintaining a max-heap with two keys. The primary key of an index is the number of connections to the currently ordered indices. The secondary key is the number of nonzeros in the corresponding $(N-1)$th-order sub-tensor, e.g., $\mathcal{X}(:, \ldots, :, i_n, :, \ldots, :)$, where the smaller secondary key values signify higher priority. The secondary keys are static. Algorithm 18 gives the pseudocode of BFS-MCS. The heap H_n is initially constructed (Line 3) according to the

secondary keys of mode-n indices. For each mode-n index (e.g., i_n) of this max-heap, BFS-MCS traverses all connected tensor indices in the modes except n, calculates the number of connections of mode-n indices, and then updates the heap using the primary and secondary keys. Let $\mathbf{m}_V$ denote a size-$|V|$ array that tracks whether a vertex has been visited. They are initialized to zeros (unvisited) and changed to 1s once after being visited. We record a mode-n index $v_n^{(0)}$, obtained from the max-heap H_n, to the permutation array perm_n. The algorithm visits all its hyperedges (i.e., all nonzeros with i_n, Line 9) and their connected and unvisited vertices from the other $(N-1)$ modes except n (Line 10). For these vertices, it again visits their hyperedges (e in Line 12) and then checks if the connected vertices in mode n (v_n) are in the heap (Line 14). If so, the primary key (connectivity) of v_n is increased by 1 and the max-heap (H_n) is updated. To summarize, this algorithm locates the sub-tensors of the neighbor vertices of $v_n^{(0)}$ to increase the primary key values of the occurred mode-n indices in H_n.

The BFS-MCS heuristic has low time complexity. We explain it using tensor terms. The innermost heap update operation (Line 15) costs $\mathcal{O}(\log(I_n))$. Each sub-tensor is only visited once with the help of the marker array $\mathbf{m}_V$. For all the sub-tensors in one tensor mode, the heap update is only performed for nnz nonzeros (hyperedges). Overall, the time complexity of BFS-MCS is $\mathcal{O}(N\,\text{nnz}\log(I_n))$ for an Nth-order tensor with nnz nonzeros, when computing perm_n for mode n.

BFS-MCS does not exactly catch the memory access pattern of an MTTKRP. It treats the contribution from the indices of all modes except n equally to the connectivity, thus the connectivity might not match the actual data reuse. Also, it uses a greedy strategy to determine the next-level vertices, which could miss the optimal global orderings, as is common to greedy heuristics for hard problems.

8.2.2 Lexi-Order

A lexical ordering of an integer vector is the standard dictionary ordering of its elements, defined as follows. Given two equal-length vectors, $\mathbf{x}$ and $\mathbf{y}$, we say $\mathbf{x} \leq \mathbf{y}$ iff either (i) all elements are the same, i.e., $\mathbf{x} = \mathbf{y}$; or (ii) there exists an index j such that $\mathbf{x}(j) < \mathbf{y}(j)$ and $\mathbf{x}(i) = \mathbf{y}(i)$ for all $0 \leq i < j$. Consider two $\{0,1\}$ vectors $\mathbf{x} = (1,0,1,0)$ and $\mathbf{y} = (1,1,0,0)$, we have $\mathbf{x} \leq \mathbf{y}$ because $\mathbf{x}(1) < \mathbf{y}(1)$ and $\mathbf{x}(i) = \mathbf{y}(i)$ for all $0 \leq i < 1$.

Lubiw [167] associates a vector $\mathbf{v}$ of size $I \cdot J$ with an $I \times J$ matrix $\mathbf{A}$, where the matrix indices (i,j) are first sorted with $i+j$ and then j. A *doubly lexical ordering* of $\mathbf{A}$ is an ordering of its rows and columns which makes $\mathbf{v}$ lexically the largest. Every real-valued matrix has a doubly lexical ordering, and such an ordering is not unique [167]. Doubly lexical ordering has a number of applications, including the efficient recognition of totally balanced, subtree, and plaid matrices and (strongly) chordal graphs. To the

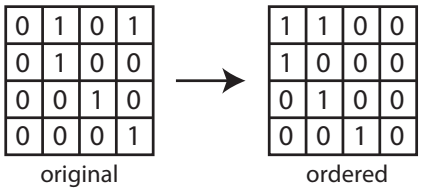


Figure 8.3: A doubly lexical ordering of a $\{0, 1\}$ matrix.

best of our knowledge, the merits of this ordering for high performance have not been investigated for sparse matrices.

We propose an iterative algorithm called `matLexiOrder` for doubly lexical ordering of matrices, where in an iteration either rows or columns are sorted lexically. Lubiw [167, Claim 2.2] shows that exchanging two rows which are lexically out of order improves the lexical ordering of the matrix. By appealing to this result, we see that the `matLexiOrder` algorithm finds a doubly lexical ordering in a finite number of iterations. This also fits well with another characterization of the doubly lexical ordering [167, 182]: a matrix is in doubly lexical ordering if both the row and column vectors are in non-increasing lexical order. A row vector is read from left to right, and a column vector is read from top to bottom. Figure 8.3 shows a doubly lexical ordering of a sample matrix. As seen in the figure, the row vectors are in non-increasing lexical order as are the column vectors.

The known doubly lexical ordering algorithms for matrices by Lubiw [167] and Paige and Tarjan [182], are "direct" ordering methods with a run time of $\mathcal{O}(\text{nnz} \log(I + J) + J)$ and $\mathcal{O}(\text{nnz} + I + J)$ space, for an $I \times J$ matrix with nnz nonzeros. We find the time complexity of these algorithms to be too high for our purpose. Furthermore, the data structures are too complex to allow an efficient generalized implementation for tensors. Since we do not aim to obtain an exact doubly lexical ordering (a close-by ordering will likely suffice to improve the MTTKRP performance), our approach will be faster while being simpler to implement.

We first describe `matLexiOrder` algorithm for a sparse matrix. This aims to show the efficiency of our iterative approach compared to others. A partition refinement technique is used to order the rows and columns alternatively. Given an ordering of the rows, the columns can be sorted lexically. This is achieved by an order preserving variant of the partition refinement method [182], which is called `orderlyRefine`. We briefly explain the partition refinement technique for ordering the columns of a matrix. Given an $I \times J$ matrix $\mathbf{A}$, all columns are initially put into a single part. Then, $\mathbf{A}$'s nonzeros are visited row by row. At a row i, each column part C is split into two parts $C_1 = C \cap \mathbf{a}(i, :)$ and $C_2 = C \setminus \mathbf{a}(i, :)$, and these two parts replace C in the order $C_1 \succ C_2$ (empty sets are discarded). Note

that this algorithm keeps the parts in a particular order which generates an ordering of the columns. `orderlyRefine` is used to refine all parts that have at least one nonzero in row i in $\mathcal{O}(|\mathbf{a}(i,:)|)$ time, where $|\mathbf{a}(i,:)|$ is the number of nonzeros of row i. Overall, `matLexiOrder` costs a linear total time of $\mathcal{O}(\text{nnz}+I+J)$ (for rows and columns ordering) per iteration and $\mathcal{O}(J)$ space. We also observe that only a small number of iterations will be enough (will be shown in Section 8.3.5 for tensors), yielding a more storage-efficient algorithm compared to the prior doubly lexical ordering methods [167, 182]. `matLexiOrder`, in particular the use of `orderlyRefine` routine a few times, is sufficient for our needs.

Algorithm 19: LEXI-ORDER for a given mode.

Data: An Nth-order sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, mode n;
Result: Permutation $perm_n$

`// Sort all nonzeros along with all but mode` n`.`
quickSort($\mathcal{X}$, coordCmp)
`// Matricize` $\mathcal{X}$ `to` $\mathbf{X}_{(n)}$`.`
$r \leftarrow$ compose (inds $([-n]), 1)$
for $m = 1, \ldots,$ nnz **do**
    $c \leftarrow \text{inds}(n, m)$`// Column index of` $\mathbf{X}_{(n)}$
    if coordCmp$(\mathcal{X}, m, m-1) = 1$ **then**
        $r \leftarrow$ compose (inds $([-n]), m)$`// Row index of` $\mathbf{X}_{(n)}$
    $\mathbf{X}_{(n)}(r, c) \leftarrow \text{val}(m)$
`// Apply partition refinement to the columns of` $\mathbf{X}_{(n)}$`.`
perm$_n \leftarrow$ `orderlyRefine` $(\mathbf{X}_{(n)})$
return perm$_n$

`// Comparison function for two indices of` $\mathcal{X}$
Function coordCmp$(\mathcal{X}, m_1, m_2)$
for $n' = 1, \ldots, N$ **do**
    if $n' \leq n$ **then**
        return -1
    if $m_1(n') > m_2(n')$ **then**
        return 1
return 0

To order tensors, we propose the LEXI-ORDER function as an extension of `matLexiOrder`. The basic idea of LEXI-ORDER is to determine the permutation of each tensor mode independently, while considering the order in other modes fixed. LEXI-ORDER sets the indices of the mode to be ordered as the columns of a matrix, the other indices as the rows and sorts the columns as described for matrices (with the order preserving partition refinement method). The precise algorithm appears in Algorithm 19, which we also illustrate in Figure 8.4 when applied to mode 1. Given a mode n, LEXI-ORDER first builds a matricized tensor in Compressed Sparse

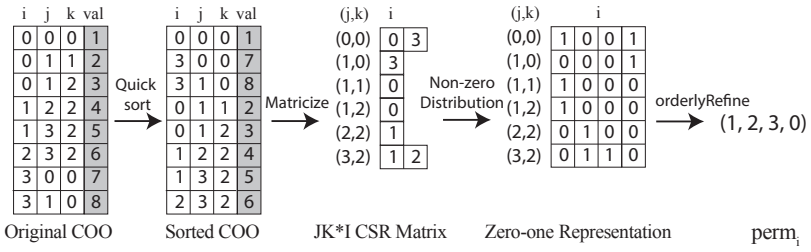


Figure 8.4: The steps of LEXI-ORDER illustrated for mode 1.

Row (CSR) sparse matrix format by a call to `quickSort` with the comparison function `coordCmp` and then by partitioning the nonzeros into the row segments (Lines 3–7). This comparison function `coordCmp` does a lexical comparison of all-but-mode-n indices, which enables efficient matricization. In other words, sorting of the tensor $\mathcal{X}$ is the same as building the matricized tensor $\mathbf{X}_{(n)}$ by rows in the fixed lexical ordering, where mode n is the column dimension and the remaining modes constitute the rows. In Figure 8.4, the sorting step orders the COO entries by (j, k) tuples, which then serve as the row indices of the matricized CSR representation of $\mathbf{X}_{(1)}$. Once the matrix is built, we construct $\{0, 1\}$ row vectors in Figure 8.4 to illustrate its nonzero distribution, which could seamlessly call `orderlyRefine` function. Apart from the `quickSort`, the other parts of LEXI-ORDER are of linear time complexity (linear in terms of tensor storage). We use OpenMP Tasks to parallelize `quickSort` to accelerate LEXI-ORDER.

Like BFS-MCS approach, LEXI-ORDER also finds the permutations for N modes of an Nth-order sparse tensor. Figure 8.5(a) illustrates the effect of LEXI-ORDER on an example $4 \times 4 \times 3$ sparse tensor. The original tensor is converted to a HICOO representation with block size $B = 2$ consisting of 5 blocks, with maximum 2 nonzeros per block. After reordering with LEXI-ORDER, the new HICOO has 3 nonzero blocks with up to 4 nonzeros per block. Thus, the blocks are denser, which should exhibit better locality behavior. However, this reordering scheme is heuristic. For example, consider Figure 8.1, we draw another HICOO representation after reordering in Figure 8.5(b). Applying LEXI-ORDER would yield a reordered HICOO representation with 4 blocks, which is the same as the input ordering, although the maximum number of nonzeros per block would increase to 4. For this tensor, LEXI-ORDER may not show a big advantage.

8.2.3 Analysis

We take the MTTKRP operation to analyze reordering behavior for COO, CSF, and HICOO formats. Recall that for a third-order sparse tensor $\mathcal{X}$, MTTKRP multiples each nonzero entry $x_{i,j,k}$ with the R-vector formed by the entry-wise product of the jth row of $\mathbf{B}$ and kth row of $\mathbf{C}$ when computing

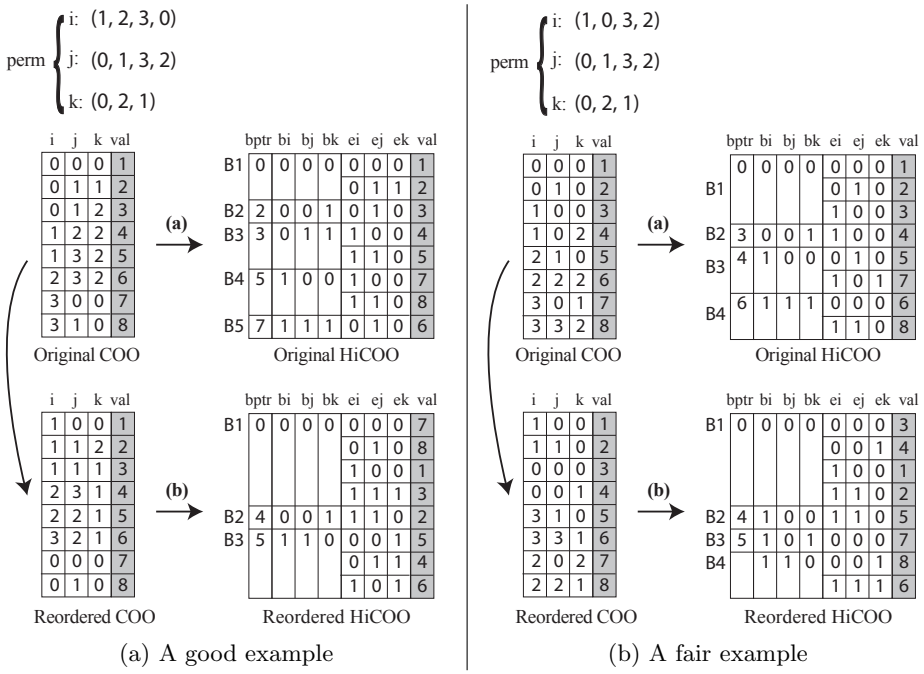


Figure 8.5: Comparison of HiCOO representations before and after Lexi-Order.

$\mathbf{M}_A$. The arithmetic intensity of Mttkrp algorithms on these formats is approximately 1/4 [155]. Thus, Mttkrp can be considered memory-bound for most computer architectures, especially CPU platforms.

In HiCOO, smaller block ratios (α_b) and larger average slice sizes per tensor block ($\overline{c_b}$) are favorable for good Mttkrp performance. Our reordering algorithms tend to closely pack nonzeros together and get denser blocks (larger $\overline{c_b}$), which potentially generates less tensor blocks (smaller α_b), thus HiCOO-Mttkrp has less memory access, more cache hits, and better performance. Moreover, smaller α_b can also reduce tensor memory requirement as a side benefit [155]. That is, for HiCOO format a reordering scheme should increase the block density, reduce the number of blocks, and increase cache locality—three related performance metrics. Reordering is more beneficial for HiCOO format than COO and CSF formats.

COO stores all nonzero indices, so relabeling does not make a difference in its storage. The same is also true for CSF; relabeling does not change the CSF's tree structure, while the order of nodes may change. For an Mttkrp with tensors stored in these two formats, the performance gain from reordering is only from the improved data locality. Though it is hard to do theoretical analysis for them, the potential better data locality could also brings performance advantages, but could be less than HiCOO's.

Tensors	Order	Dimensions	#Nnzs	Density
vast	3	$165K \times 11K \times 2$	26M	6.9×10^{-3}
nell2	3	$12K \times 9K \times 29K$	77M	2.4×10^{-5}
choa	3	$712K \times 10K \times 767$	27M	5.0×10^{-6}
darpa	3	$22K \times 22K \times 24M$	28M	2.4×10^{-9}
fb-m	3	$23M \times 23M \times 166$	100M	1.1×10^{-9}
fb-s	3	$39M \times 39M \times 532$	140M	1.7×10^{-10}
flickr	3	$320K \times 28M \times 2M$	113M	7.8×10^{-12}
deli	3	$533K \times 17M \times 3M$	140M	6.1×10^{-12}
nell1	3	$2.9M \times 2.1M \times 25M$	144M	9.1×10^{-13}
crime	4	$6K \times 24 \times 77 \times 32$	5M	1.5×10^{-2}
uber	4	$183 \times 24 \times 1140 \times 1717$	3M	3.9×10^{-4}
nips	4	$2K \times 3K \times 14K \times 17$	3M	1.8×10^{-6}
enron	4	$6K \times 6K \times 244K \times 1K$	54M	5.5×10^{-9}
flickr4d	4	$320K \times 28M \times 2M \times 731$	113M	1.1×10^{-14}
deli4d	4	$533K \times 17M \times 3M \times 1K$	140M	4.3×10^{-15}

Table 8.1: Description of sparse tensors.

8.3 Experiments

Platform. We perform experiments on a Linux-based Intel Xeon E5-2698 v3 multicore server platform with 32 physical cores distributed on two sockets, each with 2.3 GHz frequency. We only present results for sequential runs (see the paper [156] for parallel runs). The processor microarchitecture is Haswell, having 32 KiB L1 data cache and 128 GiB memory. The code artifact was written in the C language and was compiled using icc 18.0.1.

Dataset. We use the sparse tensors, derived from real-world applications, that appear in Table 8.1, ordered by decreasing nonzero density separately for third- and fourth-order tensors. Apart from choa [188], all tensors are available publicly FROSTT [203], HaTen2 [121].

Configurations. We report the results under the best configurations for the following parameters for the highest MTTKRP performance with the three formats: (i) the number of reordering iterations, and the block size B for HiCOO format; (ii) tiling or not for CSF. For HiCOO, $B = 128$ achieves the best results in most cases, and we use five reordering iterations which will be analyzed in Section 8.3.5. All experiments use approximate rank of $R = 16$. We use the total execution time of MTTKRPs in all modes for every tensor to calculate the speedup which is the ratio of the total MTTKRP time on a randomly reordered tensor over that using a specific reordering scheme. All the times are averaged over five runs.

8.3.1 COO-MTTKRP with reordering

We show the effect of the two reordering approaches on sequential COO-MTTKRP from ParTI! [154] in Figure 8.6. This COO-MTTKRP is imple-

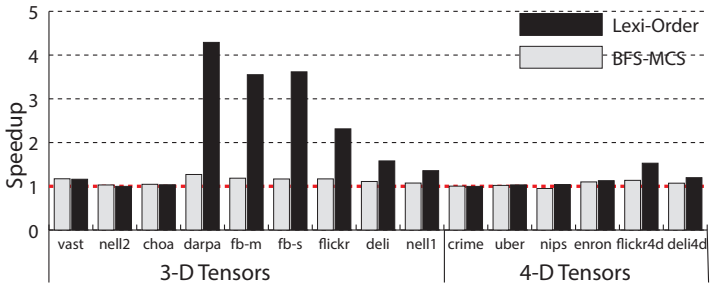


Figure 8.6: Reordered COO-MTTKRP speedup over a random reordering implementation.

mented in C using the same algorithm with TENSOR TOOLBOX [18]. For any reordering approach, after doing a BFS-MCS, LEXI-ORDER, or random reordering on the input tensor, we still sort the tensor in the mode order of $1 \succ \cdots \succ N$. Observe that LEXI-ORDER improves sequential COO-MTTKRP performance by 1.00–4.29× (1.79× on average), while BFS-MCS gets 0.95–1.27× (1.10× on average). Note that LEXI-ORDER improves the performance of COO-MTTKRP for all tensors. We conclude that this ordering is always helpful for COO-MTTKRP, while the improvements being less than what we saw for HICOO-MTTKRP.

8.3.2 CSF-MTTKRP with reordering

We show the effect of the two reordering approaches on sequential CSF-MTTKRP from SPLATT v1.1.1 [205] in Figure 8.7. CSF-MTTKRP is set to use all CSF representations (`ALLMODE`) for MTTKRPs in all modes and with tiling option on. LEXI-ORDER improves sequential CSF-MTTKRP performance by 0.65–2.33× (1.50× on average). BFS-MCS improves sequential CSF-MTTKRP performance by 1.00–1.86× (1.22× on average). Both ordering approaches improves the performance of CSF-MTTKRP on average. While BFS-MCS is always helpful in the sequential case, LEXI-ORDER is not helpful on only one tensor crime. The improvements achieved by the two reordering approaches for CSF are less than those for HICOO and COO formats. We conclude that both reordering methods are helpful for CSF-MTTKRP, but to a lesser extent than for HICOO and COO based MTTKRP.

8.3.3 HiCOO-MTTKRP with reordering

Figure 8.8 shows the speedup of the proposed reordering methods on sequential HICOO-MTTKRP. LEXI-ORDER reordering obtains 0.99–4.14× speedup (2.12× on average); while BFS-MCS reordering gets 0.99–1.88× speedup (1.34× on average). LEXI-ORDER and BFS-MCS do not behave as well on fourth-order tensors as on third-order tensors. Tensor flickr4d is

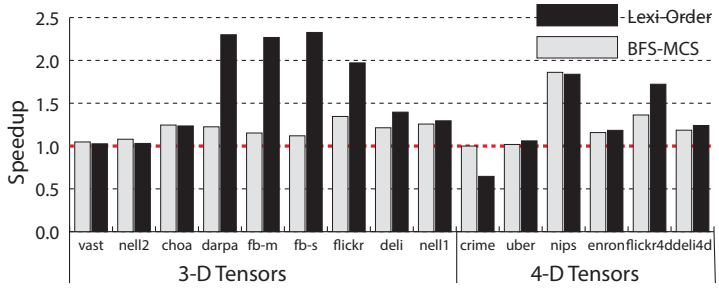


Figure 8.7: Reordered CSF-MTTKRP speedup over a random reordering implementation.

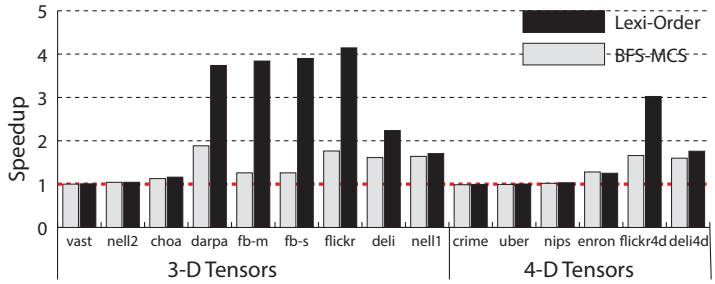


Figure 8.8: Reordered HICOO-MTTKRP speedup over a random ordering implementation.

constructed from the same data with flickr, with an extra short mode (refer to Table 8.1). LEXI-ORDER obtains 4.14× speedup on flickr while only 3.02× speedup on flickr4d. The same phenomenon is also observed on tensors deli and deli4d. This phenomenon indicates that it is harder to get good data locality on higher-order tensors, which will be justified in Table 8.2.

We investigate the two critical parameters of HICOO [155]: the block ratio (α_b) and the average slice size per tensor block ($\overline{c_b}$). Smaller α_b and larger $\overline{c_b}$ are favorable for good HICOO-MTTKRP performance. Table 8.2 lists the parameter values for all tensors before and after LEXI-ORDER, the HICOO-MTTKRP speedup (as shown in Figure 8.8), and the storage ratio of HICOO over random ordering. Generally, when both parameters α_b and $\overline{c_b}$ are largely improved, we see a good speedup and storage ratio using LEXI-ORDER. As seen for the same data in different orders, e.g., flickr4d and flickr, the α_b and $\overline{c_b}$ values after LEXI-ORDER are better in the smaller order version. This observation supports the intuition that getting good data locality is harder for higher-order tensors.

8.3.4 Reordering methods comparison

As seen above, LEXI-ORDER improves performance more than BFS-MCS in most cases. Compared to the reordering method used in SPLATT [206],

Tensors	Random reordering		Lexi-Order		Speedup		Storage
	α_b	$\overline{c_b}$	α_b	$\overline{c_b}$	seq	omp	ratio
vast	0.004	1.758	0.004	1.562	1.01	1.03	0.999
nell2	0.020	0.314	0.008	0.074	1.04	1.04	0.966
choa	0.089	0.057	0.016	0.056	1.16	1.07	0.833
darpa	0.796	0.009	0.018	0.113	3.74	1.50	0.322
fb-m	0.985	0.008	0.086	0.021	3.84	1.21	0.335
fb-s	0.982	0.008	0.099	0.020	3.90	1.23	0.336
flickr	0.999	0.008	0.097	0.025	4.14	3.66	0.277
deli	0.988	0.008	0.501	0.010	2.24	0.83	0.634
nell1	0.998	0.008	0.744	0.009	1.70	0.70	0.812
crime	0.001	37.702	0.001	8.978	0.99	1.42	1.000
uber	0.041	0.469	0.011	0.270	1.00	0.78	0.838
nips	0.016	0.434	0.004	0.435	1.03	1.34	0.921
enron	0.290	0.017	0.045	0.030	1.25	1.36	0.573
flickr4d	0.999	0.008	0.148	0.020	3.02	11.81	0.214
deli4d	0.998	0.008	0.596	0.010	1.76	1.26	0.697

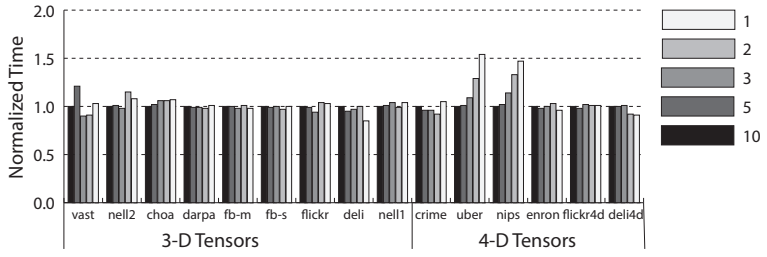
Table 8.2: HiCOO parameters before and after Lexi-Order reordering.

by setting `ALLMODE` (identical to the work [206]) to CSF-Mttkrp, BFS-MCS gets 1.04, 1.64, and 1.61× speedups on tensors nell2, nell1, and deli respectively, and Lexi-Order obtains 1.04, 1.70, and 2.24× speedups. By contrast, the speedups using graph partitioning [206] on these three tensors are 1.06, 1.11, and 1.19× and 1.06, 1.12, and 1.24× by using hypergraph partitioning [206] respectively. Our BFS-MCS and Lexi-Order schemes both outperform graph and hypergraph partitionings [206].

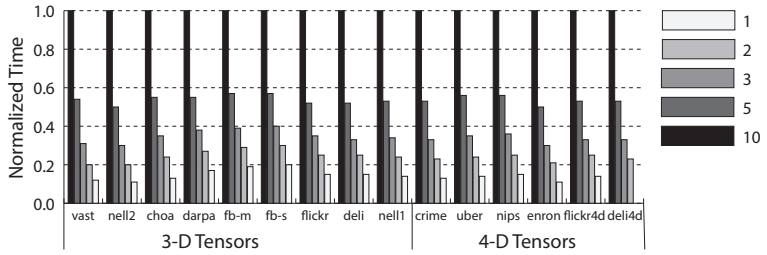
The available methods in the state-of-the-art are based on graph and hypergraph partitioning. Partitioning is a successful approach when the number of partitions is known, while the number of blocks in HiCOO is not known ahead of time. In our case, the partitions should also be ordered for better cache reuse. Additionally, partitioners are less effective for tensors than usual, as some dimensions could be very short (creating very high degree vertices). That is why the proposed ordering based methods deliver better results.

8.3.5 Effect of the number of iterations in Lexi-Order

Since Lexi-Order is iterative, we evaluate the effect of the number of iterations on HiCOO-Mttkrp performance. The results appear in Figure 8.9(a), which is normalized to the run time of 10 iterations. Mttkrp on most tensors does not vary a lot by setting different number of iterations, except vast, nell2, uber, and nips. We use 5 iterations to get good Mttkrp performance similar to that of 10 iterations, with about half of the overhead (shown in Figure 8.9(b)). But 3 or fewer iterations will get an acceptable performance when users care much about the pre-processing time.



(a) Sequential HiCOO-Mttkrp behavior



(b) The reordering overhead

Figure 8.9: The performance and overhead of different numbers of iterations.

8.4 Summary, further notes and references

Plenty recent research studied the optimization of tensor algorithms [20, 28, 46, 159, 168]. Our work sets itself apart by focusing on reordering to get better nonzero structure. Various reordering methods have been considered for matrix operations [7, 173, 190, 215].

The proposed two heuristics aim to arrange the nonzeros of a given tensor close to each other, in all modes. As said before, this would correspond to reordering the rows and columns so that all nonzeros are clustered around the diagonal in the matrix case. Heuristics based on partitioning and ordering have been used the matrix case for arranging most nonzeros around the diagonal, or diagonal blocks. An obvious alternative in the matrix case is the BFS-based, reverse Cuthill-McKee (RCM) heuristic. Given a pattern symmetric matrix $\mathbf{A}$, RCM finds a permutation (ordering) such that, the symmetrically permuted matrix $\mathbf{PAP}^T$ has nonzeros clustered around the diagonal. More precisely, RCM tries to reduce the bandwidth, which is defined as $\max\{j - i : a_{ij} \neq 0 \text{ and } j > i\}$. The RCM heuristic can be applied to pattern-wise unsymmetric, even rectangular, matrices as well. A common approach is to create a (2×2)-block matrix for a given $m \times n$ matrix $\mathbf{A}$ as

$$\mathbf{B} = \begin{pmatrix} \mathbf{I}_m & \mathbf{A} \\ \mathbf{A}^T & \mathbf{I}_n \end{pmatrix},$$

where $\mathbf{I}_m$ and $\mathbf{I}_n$ are the identity matrices if size $m \times m$ and $n \times n$, respectively. Since $\mathbf{B}$ is now pattern-symmetric, RCM can be run on it to obtain an $(m+n) \times (m+n)$ permutation matrix $\mathbf{P}$. The row and column permutations

	min	max	geomean
BFS-MCS	0.84	53.56	2.90
Lexi-Order	0.15	4.33	0.99

Table 8.3: Statistical indicators of the number of nonempty blocks obtained by BFS-MCS and Lexi-Order with respect to that of RCM on 774 matrices from the UFL collection.

for $\mathbf{A}$ can then be obtained from $\mathbf{P}$ by respecting the ordering of the first m rows and the last n rows of $\mathbf{B}$. This approach can be extended to tensors. We discuss the third order case for simplicity. Let $\mathcal{X}$ be an $I \times J \times K$ tensor. We create a matrix $\mathbf{B}$ in such a way that the first I rows/columns correspond to the first-mode indices; the following J rows/columns correspond to the second-mode indices; and the last K rows/columns correspond to the third-mode indices. Then, for each nonzero $x_{i,j,k}$ of $\mathcal{X}$, we set $b_{r,c} = 1$ whenever r and c correspond to any two indices from the set $\{i, j, k\}$. This way $\mathbf{B}$ is a symmetric matrix with ones on the main diagonal. Observe that when $\mathcal{X}$ is two dimensional (that is a matrix), we obtain the matrix $\mathbf{B}$ above. We now compare the proposed reordering heuristics with RCM.

We first compare the three methods RCM, BFS-MCS, and Lexi-Order on matrices available at the UFL collection [59]. We have tested these three heuristics on all pattern-wise unsymmetric matrices having more than 1,000 and less than 5,000,000 rows, at least three nonzeros per row and column, less than 10,000,000 nonzeros, and less than $100 \cdot \sqrt{m \times n}$ nonzeros for a matrix with m rows and n columns. At the time of experimentation, there were 774 matrices satisfying these properties. We ordered these matrices with RCM, BFS-MCS, and Lexi-Order (5 iterations), and counted the number of nonempty blocks with a block size of 128 (in each dimension). We then normalized the results of BFS-MCS and Lexi-Order with respect to that of RCM. We give the statistical indicators of these normalized results in Table 8.3. As seen in the table, RCM is better than BFS-MCS, and Lexi-Order performs as good as RCM in reducing the number of blocks.

We next compare the three methods on 11 sparse tensors from the FROSTT [203] and Haten2 repositories [121], again with a block size of 128 in each dimension. Table 8.4 shows the number of nonempty blocks obtained with these methods. In this table, the number of blocks obtained by RCM is given in absolute terms, and those obtained by BFS-MCS and Lexi-Order are given with respect to those of RCM. As seen in the table, RCM is never the best, and Lexi-Order is better than BFS-MCS for nine out of eleven cases. Overall, Lexi-Order and BFS-MCS obtain results whose geometric means are 0.48 and 0.89, respectively, of that of RCM—marking both as more effective than RCM. This is interesting, as we have seen that RCM and Lexi-Order have (nearly) the same performance in the matrix case.

	RCM	BFS-MCS	LEXI-ORDER
lbnl-network	67,208	1.11	**0.68**
nips	25,992	1.56	**0.45**
vast	113,757	1.00	**0.95**
nell-2	565,615	**0.92**	1.05
flickr	28,648,054	0.41	**0.38**
flickr4d	32,913,028	0.61	**0.51**
deli	70,691,535	0.97	**0.91**
deli4d	94,608,531	**0.81**	0.88
darpa	2,753,737	1.65	**0.19**
fb-m	53,184,625	0.66	**0.16**
fb-s	71,308,285	0.80	**0.19**
geo. mean		0.89	0.48

Table 8.4: Number of nonempty blocks obtained by RCM, BFS-MCS, and LEXI-ORDER on sparse tensors. The results of BFS-MCS and LEXI-ORDER are normalized with respect to those of RCM. For each tensor, the best result is displayed with bold.

	random	LEXI-ORDER	speedup
vast	2.02	1.60	1.26
nell-2	5.04	4.33	1.16
choa	1.78	1.43	1.25
darpa	3.18	1.48	2.15
fb-m	17.16	6.95	2.47
fb-s	25.03	9.73	2.57
flickr	10.29	5.82	1.77
deli	14.05	10.80	1.30
nell1	18.45	14.86	1.24

Table 8.5: The run time of TTM (for all modes) using a random ordering and LEXI-ORDER, and the speedup with LEXI-ORDER for sequential execution.

The proposed ordering methods are applicable to some other sparse tensor operations without any change. Among those operations, tensor-matrix multiplication (TTM) used in computing the Tucker decomposition with higher-order orthogonal iterations [61], or tensor-vector multiplications used in the higher order power method [60] are well known. The main characteristic of these operations is that the memory access of the algorithm depends on the locality of tensor indices for each dimension. If not all dimensions have the same characteristics, potentially a variant of the proposed methods in which only certain dimensions are ordered could be used. We showcase some results for the TTM case below in Table 8.5 on some of the tensors. The results are obtained on a machine with Intel Xeon CPU E5-2650v4. As seen in the table, using Lexi-Order always results in improved sequential run time for TTMs.

Recent work [46] analyses the memory access pattern of the MTTKRP operation and proposes low-level, highly effective code optimizations. The proposed approach reorganizes the code with register blocking and applies nonzero blocking in an attempt to fit rows of the factor matrices into the cache. The gain in performance remarkable. Our ordering methods are orthogonal to the mentioned and similar optimizations. For a much improved performance, we should first order the tensor using the approaches proposed in this chapter, and then further optimize the code using the low-level code optimizations.

Part IV

Closing

Chapter 9

Concluding remarks

We have presented a selection of partitioning, matching, and ordering problems in combinatorial scientific computing:

Partitioning problems: These problems arise in task decomposition for parallel computing, where load balance and low communication cost are two objectives. Depending on the task definition and the interaction of the tasks, the partitioning problems use different combinatorial models. The selected problems (Chapters 2 and 3) addressed acyclic partitioning of directed acyclic graphs and partitioning of hypergraphs. While our contributions for the former problem concerned combinatorial tools for the desired partitioning objectives and constraints, those for the second problem concerned the use of hypergraph models and associated partitioning tools for efficient tensor decomposition in the distributed memory setting.

Matching problems: These problems arise in settings where agents compete for exclusive access to resources. The most common settings include identifying nonzero diagonals in sparse matrices, routing and scheduling in data centers. A related concept is to decompose a doubly stochastic matrix as a convex combination of permutation matrices (the famous Birkhoff-von Neumann decomposition), where each permutation matrix is a perfect matching in the bipartite graph representation of the matrix. We developed approximation algorithms for matchings in graphs (Chapter 4) and effective heuristics for finding matchings in hypergraphs (Chapter 6). We also investigated (Chapter 5) the problem of finding Birkhoff-von Neumann decompositions with a small number of permutation matrices and presented complexity results and theoretical insights into the decomposition problem. On the way we generalized the decomposition to general real matrices (not only doubly stochastic) having total support.

Ordering problems: These problems arise when one wants to permute sparse matrices and tensors into desirable forms. The sought forms depend of course on the targeted applications. By focusing on a recent LU decomposition method (or on a method taking advantage of the sparsity in a novel way), we developed heuristics to permute sparse matrices into a bordered block triangular form with an aim to reduce the height of the resulting elimination tree (Chapter 7). We also investigated sparse tensor ordering problem where we sought to cluster nonzeros around the (hyper)diagonal of a sparse tensor (Chapter 8) in order to improve the performance of certain tensor operations. We proposed novel algorithms for obtaining doubly lexical ordering of sparse matrices that are simpler to implement than the known algorithms to be able to address the tensor ordering problem.

At the end of Chapters 2–8, we mentioned some immediate future work. Below, we state some more future work, which require considerable work and creativity.

Most of the presented combinatorial algorithms come with applications. Others are made applicable by practical and efficient implementations; further developments on the applications' side are required for these to be used in practice. In particular, the use of acyclic partitioner (the problem of Chapter 2) in a parallel system, with the help of a runtime system, requires the whole graph to be available. This is usually not the case; an autotuning like approach in which one first creates a task graph by a cold run and then partitions this graph to determine a schedule can prove useful. In a similar vein, we mentioned the lack of proper software implementing the LU factorization method using elimination trees at the end of Chapter 7. Such an implementation will give rise to a number of combinatorial problems (e.g., the peak memory requirement in factoring a BBT ordered matrix with elimination trees), and seems to be a new play field.

The original Karp-Sipser heuristic for the cardinality matching problem applies degree-1 and degree-2 reduction rules. The first rule is well implemented in practical settings (see Chapters 4 and 6). On the other hard, fast implementations of the degree-2 reduction rule for undirected graphs are sought. A straightforward implementation of this rule can result in the worst case $\Omega(n^2)$ total time for a graph with n vertices. This is obviously too much to afford in theory for general sparse graphs with $O(n)$ or $O(n \log n)$ edges. Are there worst-case linear time algorithms for implementing this rule in the context of the Karp-Sipser heuristic?

The Birkhoff-von Neumann decomposition (Chapter 5) has well founded applications in routing. Our use of this decomposition in solving linear systems is a first attempt in making it useful for numerical algorithms. In the proposed approach, the cost of constructing a preconditioner with a few terms from a BvN decomposition is equivalent to solving a few maxi-

mum bottleneck matching problem—this can be costly in practical settings. Can we use a relaxed definition of the BvN decomposition in which sub-permutation matrices are allowed (such decompositions are used in routing problems) and make related preconditioner more practical while retaining numerical properties?

Bibliography

[1] IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/, 2010. [Cited on pp. 96, 121]

[2] Evrim Acar, Daniel M. Dunlavy, and Tamara G. Kolda. A scalable optimization approach for fitting canonical tensor decompositions. *Journal of Chemometrics*, 25(2):67–86, 2011. [Cited on pp. 11, 37]

[3] Seher Acer, Tugba Torun, and Cevdet Aykanat. Improving medium-grain partitioning for scalable sparse tensor decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 29(12):2814–2825, Dec 2018. [Cited on pp. 52]

[4] Kunal Agrawal, Jeremy T. Fineman, Jordan Krage, Charles E. Leiserson, and Sivan Toledo. Cache-conscious scheduling of streaming applications. In *Proc. Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 236–245, New York, NY, USA, 2012. ACM. [Cited on pp. 4]

[5] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Multifrontal QR factorization for multicore architectures over runtime systems. In *Euro-Par 2013 Parallel Processing*, LNCS, pages 521–532. Springer Berlin Heidelberg, 2013. [Cited on pp. 131]

[6] Ron Aharoni and Penny Haxell. Hall's theorem for hypergraphs. *Journal of Graph Theory*, 35(2):83–88, 2000. [Cited on pp. 119]

[7] Kadir Akbudak and Cevdet Aykanat. Exploiting locality in sparse matrix-matrix multiplication on many-core architectures. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2258–2271, Aug 2017. [Cited on pp. 160]

[8] Patrick R. Amestoy, Iain S. Duff, and Jean-Yves L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods in Applied Mechanics and Engineering*, 184(2–4):501–520, 2000. [Cited on pp. 131, 132]

[9] Patrick R. Amestoy, Iain S. Duff, Daniel Ruiz, and Bora Uçar. A parallel matrix scaling algorithm. In J. M. Palma, P. R. Amestoy, M. Daydé, M. Mattoso, and J. C. Lopes, editors, *8th International Conference on High Performance Computing for Computational Science (VECPAR):*, volume 5336, pages 301–313. Springer Berlin Heidelberg, 2008. [Cited on pp. 56]

[10] Michael Anastos and Alan Frieze. Finding perfect matchings in random cubic graphs in linear time. *arXiv preprint arXiv:1808.00825*, 2018. [Cited on pp. 125]

[11] Claus A. Andersson and Rasmus Bro. The N-way toolbox for MATLAB. *Chemometrics and Intelligent Laboratory Systems*, 52(1):1–4, 2000. [Cited on pp. 11, 37]

[12] Hartwig Anzt, Edmond Chow, and Jack Dongarra. Iterative sparse triangular solves for preconditioning. In Larsson Jesper Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015 Parallel Processing*, pages 650–661. Springer Berlin Heidelberg, 2015. [Cited on pp. 107]

[13] Jonathan Aronson, Martin Dyer, Alan Frieze, and Stephen Suen. Randomized greedy matching II. *Random Structures & Algorithms*, 6(1):55–73, 1995. [Cited on pp. 58]

[14] Jonathan Aronson, Alan Frieze, and Boris G. Pittel. Maximum matchings in sparse random graphs: Karp-Sipser revisited. *Random Structures & Algorithms*, 12(2):111–177, 1998. [Cited on pp. 56, 58]

[15] Cevdet Aykanat, Berkant B. Cambazoglu, and Bora Uçar. Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68:609–625, 2008. [Cited on pp. 46]

[16] Ariful Azad, Mahantesh Halappanavar, Sivasankaran Rajamanickam, Erik G. Boman, Arif Khan, and Alex Pothen. Multithreaded algorithms for maximum matching in bipartite graphs. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, pages 860–872, Shanghai, China, 2012. [Cited on pp. 57, 59, 66, 73, 74]

[17] Brett W. Bader and Tamara G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, December 2007. [Cited on pp. 11, 37]

[18] Brett W. Bader, Tamara G. Kolda, et al. Matlab tensor toolbox version 3.0. Available online `http://www.sandia.gov/~tgkolda/TensorToolbox/`, 2017. [Cited on pp. 11, 37, 38, 157]

[19] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013. [Cited on pp. 69]

[20] Muthu Baskaran, Tom Henretty, Benoit Pradelle, M. Harper Langston, David Bruns-Smith, James Ezick, and Richard Lethin. Memory-efficient parallel tensor decompositions. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sep. 2017. [Cited on pp. 160]

[21] James Bennett and Stan Lanning. The Netflix Prize. In *Proceedings of KDD cup and workshop*, page 35, 2007. [Cited on pp. 10, 48]

[22] Anne Benoit, Yves Robert, and Frédéric Vivien. *A Guide to Algorithm Design: Paradigms, Methods, and Complexity Analysis*. CRC Press, Boca Raton, FL, 2014. [Cited on pp. 93]

[23] Michele Benzi and Bora Uçar. Preconditioning techniques based on the Birkhoff–von Neumann decomposition. *Computational Methods in Applied Mathematics*, 17:201–215, 2017. [Cited on pp. 106, 108]

[24] Piotr Berman and Marek Karpinski. Improved approximation lower bounds on small occurence optimization. *ECCC Report*, 2003. [Cited on pp. 113]

[25] Garrett Birkhoff. Tres observaciones sobre el algebra lineal. *Universidad Nacional de Tucumán, Series A*, 5:147–154, 1946. [Cited on pp. 8]

[26] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. Efficient parallel and external matching. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, pages 659–670. Springer Berlin Heidelberg, 2013. [Cited on pp. 59]

[27] Rob H. Bisseling and Wouter Meesen. Communication balancing in parallel sparse matrix-vector multiplication. *Electronic Transactions on Numerical Analysis*, 21:47–65, 2005. [Cited on pp. 4, 47]

[28] Zachary Blanco, Bangtian Liu, and Maryam Mehri Dehnavi. CSTF: Large-scale sparse tensor factorizations on distributed platforms. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 21:1–21:10, New York, NY, USA, 2018. ACM. [Cited on pp. 160]

[29] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 308–317. ACM, 2012. [Cited on pp. 59, 73, 74, 75]

[30] Norbert Blum. A new approach to maximum matching in general graphs. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, pages 586–597, London, UK, 1990. Springer-Verlag. [Cited on pp. 6, 76]

[31] Richard A. Brualdi. The diagonal hypergraph of a matrix (bipartite graph). *Discrete Mathematics*, 27(2):127–147, 1979. [Cited on pp. 92]

[32] Richard A. Brualdi. Notes on the Birkhoff algorithm for doubly stochastic matrices. *Canadian Mathematical Bulletin*, 25(2):191–199, 1982. [Cited on pp. 9, 92, 94, 98]

[33] Richard A. Brualdi and Peter M. Gibson. Convex polyhedra of doubly stochastic matrices: I. Applications of the permanent function. *Journal of Combinatorial Theory, Series A*, 22(2):194–230, 1977. [Cited on pp. 9]

[34] Richard A. Brualdi and Herbert J. Ryser. *Combinatorial Matrix Theory*, volume 39 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, UK; New York, USA; Melbourne, Australia, 1991. [Cited on pp. 8]

[35] Thang Nguyen Bui and Curt Jones. A heuristic for reducing fill-in in sparse matrix factorization. In *Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing*, pages 445–452. SIAM, 1993. [Cited on pp. 4, 15]

[36] Peter J. Cameron. Notes on combinatorics. `https://cameroncounts.files.wordpress.com/2013/11/comb.pdf`, (last checked Jan. 2019) 2013. [Cited on pp. 76]

[37] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka Jr., and Tom M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, volume 5, page 3, 2010. [Cited on pp. 48]

[38] J. Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition. *Psychometrika*, 35(3):283–319, 1970. [Cited on pp. 36]

[39] Ümit V. Çatalyürek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, Jul 1999. [Cited on pp. 41, 140]

[40] Ümit V. Çatalyürek and Cevdet Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0.* Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at `http://bmi.osu.edu/~umit/software.htm`, 1999. [Cited on pp. 4, 7, 25, 47, 143]

[41] Ümit V. Çatalyürek, Kamer Kaya, and Bora Uçar. On shared-memory parallelization of a sparse matrix scaling algorithm. In *2012 41st International Conference on Parallel Processing*, pages 68–77, Los Alamitos, CA, USA, September 2012. IEEE Computer Society. [Cited on pp. 56]

[42] Ümit V. Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora Uçar. Multithreaded clustering for multi-level hypergraph partitioning. In *26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 848–859, Shanghai, China, 2012. [Cited on pp. 59]

[43] Cheng-Shang Chang, Wen-Jyh Chen, and Hsiang-Yi Huang. On service guarantees for input-buffered crossbar switches: A capacity decomposition approach by Birkhoff and von Neumann. In *Quality of Service, 1999. IWQoS '99. 1999 Seventh International Workshop on*, pages 79–86, 1999. [Cited on pp. 8]

[44] Jianer Chen, Yang Liu, Songjian Lu, Barry O'Sullivan, and Igor Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. *Journal of the ACM*, 55(5):21:1–21:19, 2008. [Cited on pp. 142]

[45] Joseph Cheriyan. Randomized $\tilde{O}(M(|V|))$ algorithms for problems in matching theory. *SIAM Journal on Computing*, 26(6):1635–1655, 1997. [Cited on pp. 76]

[46] Jee Choi, Xing Liu, Shaden Smith, and Tyler Simon. Blocking optimization techniques for sparse tensor computation. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 568–577, Vancouver, British Columbia, Canada, 2018. [Cited on pp. 160, 163]

[47] Joon Hee Choi and S. V. N. Vishwanathan. DFacTo: Distributed factorization of tensors. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *27th Advances in Neural Information Processing Systems*, pages 1296–1304. Curran Associates, Inc., Montreal, Quebec, Canada, 2014. [Cited on pp. 11, 37, 38, 39]

[48] Edmond Chow and Aftab Patel. Fine-grained parallel incomplete LU factorization. *SIAM Journal on Scientific Computing*, 37(2):C169–C193, 2015. [Cited on pp. 107]

[49] Andrzej Cichocki, Danilo P. Mandic, Lieven De Lathauwer, Guoxu Zhou, Qibin Zhao, Cesar Caiafa, and Anh-Huy Phan. Tensor decompositions for signal processing applications: From two-way to multiway component analysis. *IEEE Signal Processing Magazine*, 32(2):145–163, March 2015. [Cited on pp. 10]

[50] Edith Cohen. Structure prediction and computation of sparse matrix products. *Journal of Combinatorial Optimization*, 2(4):307–332, Dec 1998. [Cited on pp. 127]

[51] Thomas F. Coleman and Wei Xu. Parallelism in structured Newton computations. In *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007*, pages 295–302, Forschungszentrum Jülich and RWTH Aachen University, Germany, 2007. [Cited on pp. 3]

[52] Thomas F. Coleman and Wei Xu. Fast (structured) Newton computations. *SIAM Journal on Scientific Computing*, 31(2):1175–1191, 2009. [Cited on pp. 3]

[53] Thomas F. Coleman and Wei Xu. *Automatic Differentiation in MATLAB using ADMAT with Applications*. SIAM, 2016. [Cited on pp. 3]

[54] Jason Cong, Zheng Li, and Rajive Bagrodia. Acyclic multi-way partitioning of Boolean networks. In *Proceedings of the 31st Annual Design Automation Conference*, DAC'94, pages 670–675, New York, NY, USA, 1994. ACM. [Cited on pp. 4, 17]

[55] Elizabeth H. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th national conference*, pages 157–172, New York, NY, USA, 1969. ACM. [Cited on pp. 12]

[56] Marek Cygan. Improved approximation for 3-dimensional matching via bounded pathwidth local search. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 509–518. IEEE, 2013. [Cited on pp. 113, 120]

[57] Marek Cygan, Fabrizio Grandoni, and Monaldo Mastrolilli. How to sell hyperedges: The hypermatching assignment problem. In *Proc. of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 342–351. SIAM, 2013. [Cited on pp. 113]

[58] Joseph Czyzyk, Michael P. Mesnier, and Jorge J. Moré. The NEOS Server. *IEEE Journal on Computational Science and Engineering,*, 5(3):68–75, 1998. [Cited on pp. 96]

[59] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011. [Cited on pp. 25, 69, 82, 105, 143, 161]

[60] Lieven De Lathauwer, Pierre Comon, Bart De Moor, and Joos Vandewalle. Higher-order power method–Application in independent component analysis. In *Proceedings NOLTA'95*, pages 91–96, Las Vegas, USA, 1995. [Cited on pp. 163]

[61] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. On the best rank-1 and rank-$(R_1, R_2, \ldots, R_N)$ approximation of higher-order tensors. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1324–1342, 2000. [Cited on pp. 163]

[62] Dominique de Werra. Variations on the Theorem of Birkhoff–von Neumann and extensions. *Graphs and Combinatorics*, 19(2):263–278, Jun 2003. [Cited on pp. 108]

[63] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999. [Cited on pp. 132]

[64] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, 1999. [Cited on pp. 132]

[65] Mehmet Deveci, Kamer Kaya, Ümit V. Çatalyürek, and Bora Uçar. A push-relabel-based maximum cardinality matching algorithm on GPUs. In *42nd International Conference on Parallel Processing*, pages 21–29, Lyon, France, 2013. [Cited on pp. 57]

[66] Mehmet Deveci, Kamer Kaya, Bora Uçar, and Ümit V. Çatalyürek. GPU accelerated maximum cardinality matching algorithms for bipartite graphs. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *LNCS*, pages 850–861. Springer Berlin Heidelberg, 2013. [Cited on pp. 57]

[67] Pat Devlin and Jeff Kahn. Perfect fractional matchings in k-out hypergraphs. *arXiv preprint arXiv:1703.03513*, 2017. [Cited on pp. 114, 121]

[68] Elizabeth D. Dolan. The NEOS Server 4.0 administrative guide. Technical Memorandum ANL/MCS-TM-250, Mathematics and Computer Science Division, Argonne National Laboratory, 2001. [Cited on pp. 96]

[69] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002. [Cited on pp. 26]

[70] Jack J. Dongarra, Fred G. Gustavson, and Alan H. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):pp. 91–112, 1984. [Cited on pp. 134]

[71] Iain S. Duff, Kamer Kaya, and Bora Uçar. Design, implementation, and analysis of maximum transversal algorithms. *ACM Transactions on Mathematical Software*, 38(2):13:1–13:31, 2011. [Cited on pp. 6, 56, 58]

[72] Iain S. Duff and Jacko Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22:973–996, 2001. [Cited on pp. 98, 143]

[73] Fanny Dufossé, Kamer Kaya, Ioannis Panagiotas, and Bora Uçar. Approximation algorithms for maximum matchings in undirected graphs. In *2018 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*, pages 56–65, 2018. [Cited on pp. 75, 79, 113]

[74] Fanny Dufossé, Kamer Kaya, Ioannis Panagiotas, and Bora Uçar. Further notes on Birkhoff–von Neumann decomposition of doubly stochastic matrices. *Linear Algebra and its Applications*, 554:68–78, 2018. [Cited on pp. 101, 113, 117]

[75] Fanny Dufossé, Kamer Kaya, Ioannis Panagiotas, and Bora Uçar. Effective heuristics for matchings in hypergraphs. Research Report RR-9224, Inria Grenoble Rhône-Alpes, November 2018. (will appear in SEA^2). [Cited on pp. 118, 119, 122]

[76] Fanny Dufossé, Kamer Kaya, and Bora Uçar. Two approximation algorithms for bipartite matching on multicore architectures. *Journal of Parallel and Distributed Computing*, 85:62–78, 2015. IPDPS 2014 Selected Papers on Numerical and Combinatorial Algorithms. [Cited on pp. 56, 60, 65, 66, 67, 69, 77, 79, 81, 113, 117, 125]

[77] Andrew Lloyd Dulmage and Nathan Saul Mendelsohn. Coverings of bipartite graphs. *Canadian Journal of Mathematics*, 10:517–534, 1958. [Cited on pp. 68, 143]

[78] Martin E. Dyer and Alan M. Frieze. Randomized greedy matching. *Random Structures & Algorithms*, 2(1):29–45, 1991. [Cited on pp. 58, 113, 115]

[79] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965. [Cited on pp. 76]

[80] Lawrence C. Eggan. Transition graphs and the star-height of regular events. *The Michigan Mathematical Journal*, 10(4):385–397, 1963. [Cited on pp. 5]

[81] Stanley C. Eisenstat and Joseph W. H. Liu. The theory of elimination trees for sparse unsymmetric matrices. *SIAM Journal on Matrix Analysis and Applications*, 26(3):686–705, 2005. [Cited on pp. 5, 131, 132, 133, 135]

[82] Stanley C. Eisenstat and Joseph W. H. Liu. A tree-based dataflow model for the unsymmetric multifrontal method. *Electronic Transactions on Numerical Analysis*, 21:1–19, 2005. [Cited on pp. 132]

[83] Stanley C. Eisenstat and Joseph W. H. Liu. Algorithmic aspects of elimination trees for sparse unsymmetric matrices. *SIAM Journal on Matrix Analysis and Applications*, 29(4):1363–1381, 2008. [Cited on pp. 5, 132, 143, 145]

[84] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. On characterizing the data access complexity of programs. *ACM SIGPLAN Notices - POPL'15*, 50(1):567–580, January 2015. [Cited on pp. 3]

[85] Paul Erdös and Alfréd Rényi. On random matrices. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 8(3):455–461, 1964. [Cited on pp. 72]

[86] Bastiaan Onne Fagginger Auer and Rob H. Bisseling. A GPU algorithm for greedy graph matching. In *Facing the Multicore Challenge II*, volume 7174 of *LNCS*, pages 108–119. Springer-Verlag Berlin, 2012. [Cited on pp. 59]

[87] Naznin Fauzia, Venmugil Elango, Mahesh Ravishankar, J. Ramanujam, Fabrice Rastello, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential. *ACM Transactions on Architecture and Code Optimization*, 10(4):53:1–53:29, December 2013. [Cited on pp. 3, 16]

[88] Charles M. Fiduccia and Robert M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th Design Automation Conference*, pages 175–181, Las Vegas, USA, 1982. IEEE. [Cited on pp. 17, 22]

[89] Joel Franklin and Jens Lorenz. On the scaling of multidimensional matrices. *Linear Algebra and its Applications*, 114:717–735, 1989. [Cited on pp. 114]

[90] Alan M. Frieze. Maximum matchings in a class of random graphs. *Journal of Combinatorial Theory, Series B*, 40(2):196–212, 1986. [Cited on pp. 76, 82, 121]

[91] Aurélien Froger, Olivier Guyon, and Eric Pinson. A set packing approach for scheduling passenger train drivers: the French experience. In *RailTokyo2015*, Tokyo, Japan, March 2015. [Cited on pp. 7]

[92] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989. [Cited on pp. 6, 76]

[93] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. [Cited on pp. 3, 47, 93]

[94] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973. [Cited on pp. 136]

[95] Alan J. George. *Computer implementation of the finite element method*. PhD thesis, Stanford University, Stanford, CA, USA, 1971. [Cited on pp. 12]

[96] Andrew V. Goldberg and Robert Kennedy. Global price updates help. *SIAM Journal on Discrete Mathematics*, 10(4):551–572, 1997. [Cited on pp. 6]

[97] Olaf Görlitz, Sergej Sizov, and Steffen Staab. Pints: Peer-to-peer infrastructure for tagging systems. In *Proceedings of the 7th International Conference on Peer-to-peer Systems*, IPTPS'08, page 19, Berkeley, CA, USA, 2008. USENIX Association. [Cited on pp. 48]

[98] Georg Gottlob and Gianluigi Greco. Decomposing combinatorial auctions and set packing problems. *Journal of the ACM*, 60(4):24:1–24:39, September 2013. [Cited on pp. 7]

[99] Lars Grasedyck. Hierarchical singular value decomposition of tensors. *SIAM Journal on Matrix Analysis and Applications*, 31(4):2029–2054, 2010. [Cited on pp. 52]

[100] William Gropp and Jorge J. Moré. Optimization environments and the NEOS Server. In Martin D. Buhman and Arieh Iserles, editors, *Approximation Theory and Optimization*, pages 167–182. Cambridge University Press, 1997. [Cited on pp. 96]

[101] Hermann Gruber. Digraph complexity measures and applications in formal language theory. *Discrete Mathematics & Theoretical Computer Science*, 14(2):189–204, 2012. [Cited on pp. 5]

[102] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010. [Cited on pp. 47]

[103] Abdou Guermouche, Jean-Yves L'Excellent, and Gil Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9):1191–1218, 2003. [Cited on pp. 131]

[104] Anshul Gupta. Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 24(2):529–552, 2002. [Cited on pp. 132]

[105] Mahantesh Halappanavar, John Feo, Oreste Villa, Antonino Tumeo, and Alex Pothen. Approximate weighted matching on emerging manycore and multithreaded architectures. *International Journal of High Performance Computing Applications*, 26(4):413–430, 2012. [Cited on pp. 59]

[106] Magnús M. Halldórsson. Approximating discrete collections via local improvements. In *The Sixth annual ACM-SIAM symposium on Discrete Algorithms (SODA)*, volume 95, pages 160–169, San Francisco, California, USA, 1995. SIAM. [Cited on pp. 113]

[107] Richard A. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multi-modal factor analysis. *UCLA Working Papers in Phonetics*, 16:1–84, 1970. [Cited on pp. 36]

[108] Nicholas J. A. Harvey. Algebraic algorithms for matching and matroid problems. *SIAM Journal on Computing*, 39(2):679–702, 2009. [Cited on pp. 76]

[109] Elad Hazan, Shmuel Safra, and Oded Schwartz. On the complexity of approximating k-dimensional matching. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 83–97. Springer, 2003. [Cited on pp. 113]

[110] Elad Hazan, Shmuel Safra, and Oded Schwartz. On the complexity of approximating k-set packing. *Computational Complexity*, 15(1):20–39, 2006. [Cited on pp. 7]

[111] Bruce Hendrickson. Load balancing fictions, falsehoods and fallacies. *Applied Mathematical Modelling*, 25:99–108, 2000. [Cited on pp. 41]

[112] Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000. [Cited on pp. 41]

[113] Bruce Hendrickson and Robert Leland. The Chaco user's guide, version 1.0. Technical Report SAND93–2339, Sandia National Laboratories, Albuquerque, NM, October 1993. [Cited on pp. 4, 15, 16]

[114] Julien Herrmann, Jonathan Kho, Bora Uçar, Kamer Kaya, and Ümit V. Çatalyürek. Acyclic partitioning of large directed acyclic graphs. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID*, pages 371–380, Madrid, Spain, May 2017. [Cited on pp. 15, 16, 17, 18, 28, 30]

[115] Julien Herrmann, M. Yusuf Özkaya, Bora Uçar, Kamer Kaya, and Ümit V. Çatalyürek. Multilevel algorithms for acyclic partitioning of directed acyclic graphs. *SIAM Journal on Scientific Computing*, 2019. to appear. [Cited on pp. 19, 20, 26, 29, 31]

[116] Jonathan D. Hogg, John K. Reid, and Jennifer A. Scott. Design of a multicore sparse Cholesky factorization using DAGs. *SIAM Journal on Scientific Computing*, 32(6):3627–3649, 2010. [Cited on pp. 131]

[117] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973. [Cited on pp. 6, 62]

[118] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, New York, 1985. [Cited on pp. 109]

[119] Cor A. J. Hurkens and Alexander Schrijver. On the size of systems of sets every t of which have an SDR, with an application to the worst-case ratio of heuristics for packing problems. *SIAM Journal on Discrete Mathematics*, 2(1):68–72, 1989. [Cited on pp. 113, 120]

[120] Oscar H. Ibarra and Shlomo Moran. Deterministic and probabilistic algorithms for maximum bipartite matching via fast matrix multiplication. *Information Processing Letters*, pages 12–15, 1981. [Cited on pp. 76]

[121] Inah Jeon, Evangelos E. Papalexakis, U Kang, and Christos Faloutsos. Haten2: Billion-scale tensor decompositions. In *IEEE 31st International Conference on Data Engineering (ICDE)*, pages 1047–1058, April 2015. [Cited on pp. 156, 161]

[122] Jochen A. G. Jess and Hendrikus Gerardus Maria Kees. A data structure for parallel L/U decomposition. *IEEE Transactions on Computers*, 31(3):231–239, 1982. [Cited on pp. 134]

[123] U Kang, Evangelos Papalexakis, Abhay Harpale, and Christos Faloutsos. GigaTensor: Scaling tensor analysis up by 100 times - Algorithms and discoveries. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 316–324, New York, NY, USA, 2012. ACM. [Cited on pp. 10, 11, 37]

[124] Michał Karoński, Ed Overman, and Boris Pittel. On a perfect matching in a random bipartite digraph with average out-degree below two. *arXiv e-prints*, page arXiv:1903.05764, Mar 2019. [Cited on pp. 59, 121]

[125] Michał Karoński and Boris Pittel. Existence of a perfect matching in a random $(1 + e^{-1})$–out bipartite graph. *Journal of Combinatorial Theory, Series B*, 88:1–16, May 2003. [Cited on pp. 59, 67, 81, 82, 121]

[126] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposiax, pages 85–103. Springer US, 1972. [Cited on pp. 7, 142]

[127] Richard M. Karp, Alexander H. G. Rinnooy Kan, and Rakesh V. Vohra. Average case analysis of a heuristic for the assignment problem. *Mathematics of Operations Research*, 19:513–522, 1994. [Cited on pp. 81]

[128] Richard M. Karp and Michael Sipser. Maximum matching in sparse random graphs. In *22nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 364–375, Nashville, TN, USA, 1981. [Cited on pp. 56, 58, 67, 75, 81, 113]

[129] Richard M. Karp, Umesh V. Vazirani, and Vijay V. Vazirani. An optimal algorithm for on-line bipartite matching. In *22nd annual ACM symposium on Theory of computing (STOC)*, pages 352–358, Baltimore, MD, USA, 1990. [Cited on pp. 57]

[130] George Karypis and Vipin Kumar. *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*. University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Cent., Minneapolis, 1998. [Cited on pp. 4, 16, 23, 27, 142]

[131] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint hypergraph partitioning. Technical Report 99-034, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 55455, November 1998. [Cited on pp. 34]

[132] Kamer Kaya, Johannes Langguth, Fredrik Manne, and Bora Uçar. Push-relabel based algorithms for the maximum transversal problem. *Computers and Operations Research*, 40(5):1266–1275, 2013. [Cited on pp. 6, 56, 58]

[133] Kamer Kaya and Bora Uçar. Constructing elimination trees for sparse unsymmetric matrices. *SIAM Journal on Matrix Analysis and Applications*, 34(2):345–354, 2013. [Cited on pp. 5]

[134] Oguz Kaya and Bora Uçar. Scalable sparse tensor decompositions in distributed memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 77:1–77:11, Austin, Texas, 2015. ACM, New York, NY, USA. [Cited on pp. 7]

[135] Oguz Kaya and Bora Uçar. Parallel Candecomp/Parafac decomposition of sparse tensors using dimension trees. *SIAM Journal on Scientific Computing*, 40(1):C99–C130, 2018. [Cited on pp. 52]

[136] Enver Kayaaslan and Bora Uçar. Reducing elimination tree height for parallel LU factorization of sparse unsymmetric matrices. In *21st International Conference on High Performance Computing (HiPC 2014)*, pages 1–10, Goa, India, December 2014. [Cited on pp. 136]

[137] Brian W. Kernighan. Optimal sequential partitions of graphs. *Journal of the ACM*, 18(1):34–40, January 1971. [Cited on pp. 16]

[138] Shiva Kintali, Nishad Kothari, and Akash Kumar. Approximation algorithms for directed width parameters. *CoRR*, abs/1107.4824, 2011. [Cited on pp. 138]

[139] Philip A. Knight. The Sinkhorn–Knopp algorithm: Convergence and applications. *SIAM Journal on Matrix Analysis and Applications*, 30(1):261–275, 2008. [Cited on pp. 55, 68]

[140] Philip A. Knight and Daniel Ruiz. A fast algorithm for matrix balancing. *IMA Journal of Numerical Analysis*, 33(3):1029–1047, 2013. [Cited on pp. 56, 79]

[141] Philip A. Knight, Daniel Ruiz, and Bora Uçar. A symmetry preserving algorithm for matrix scaling. *SIAM Journal on Matrix Analysis and Applications*, 35(3):931–955, 2014. [Cited on pp. 55, 56, 68, 79, 105]

[142] Tamara G. Kolda and Brett Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009. [Cited on pp. 10, 37]

[143] Tjalling C. Koopmans and Martin Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25(1):53–76, 1957. [Cited on pp. 109]

[144] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, and James Avery. Fusion of parallel array operations. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 71–85, New York, NY, USA, 2016. ACM. [Cited on pp. 3]

[145] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: A virtual machine approach to portable parallelism. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 312–321, Washington, DC, USA, 2014. [Cited on pp. 3]

[146] Janardhan Kulkarni, Euiwoong Lee, and Mohit Singh. Minimum Birkhoff-von Neumann decomposition. Preliminary version `http://www.cs.cmu.edu/~euiwoonl/sparsebvn.pdf` of the paper which appeared in Proc. 19th International Conference Integer Programming and Combinatorial Optimization (IPCO 2017), Waterloo, ON, Canada, pp. 343–354, June 2017. [Cited on pp. 9]

[147] Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. Finding Near-Optimal Independent Sets at Scale. In *Proceedings of the 18th Workshop on Algorithm Engineering and Experiments (ALENEX'16)*, pages 138–150, Arlington, Virginia, USA, 2016. SIAM. [Cited on pp. 114, 127]

[148] Johannes Langguth, Fredrik Manne, and Peter Sanders. Heuristic initialization for bipartite matching problems. *Journal of Experimental Algorithmics*, 15:1.1–1.22, 2010. [Cited on pp. 6, 56, 58]

[149] Yanzhe (Murray) Lei, Stefanus Jasin, Joline Uichanco, and Andrew Vakhutinsky. Randomized product display (ranking), pricing, and order fulfillment for e-commerce retailers. *SSRN Electronic Journal*, 2018. [Cited on pp. 9]

[150] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley–Teubner, Chichester, U.K., 1990. [Cited on pp. 34]

[151] Renaud Lepère and Denis Trystram. A new clustering algorithm for large communication delays. In *16th International Parallel and Distributed Processing Symposium (IPDPS)*, page (CDROM), Fort Lauderdale, FL, USA, 2002. IEEE Computer Society. [Cited on pp. 32]

[152] Hanoch Levy and David W. Low. A contraction algorithm for finding small cycle cutsets. *Journal of Algorithms*, 9(4):470–493, 1988. [Cited on pp. 142]

[153] Jiajia Li, Jee Choi, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. Model-driven sparse CP decomposition for higher-order tensors. In *IPDPS 2017, 31th IEEE International Symposium on Parallel and Distributed Processing*, pages 1048–1057, Orlando, FL, USA, May 2017. [Cited on pp. 52]

[154] Jiajia Li, Yuchen Ma, and Richard Vuduc. ParTI!: A Parallel Tensor Infrastructure for Multicore CPU and GPUs (Version 0.1.0). Available from `https://github.com/hpcgarage/ParTI`, 2016. [Cited on pp. 156]

[155] Jiajia Li, Jimeng Sun, and Richard Vuduc. HiCOO: Hierarchical storage of sparse tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'18, New York, NY, USA, 2018. [Cited on pp. 148, 149, 155, 158]

[156] Jiajia Li, Bora Uçar, Ümit V. Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. Efficient and effective sparse tensor reordering. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 227–237, New York, NY, USA, 2019. ACM. [Cited on pp. 147, 156]

[157] Xiaoye S. Li and James W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Proc. of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–17, Washington, DC, USA, 1998. [Cited on pp. 132]

[158] Athanasios P. Liavas and Nicholas D. Sidiropoulos. Parallel algorithms for constrained tensor factorization via alternating direction method of multipliers. *IEEE Transactions on Signal Processing*, 63(20):5450–5463, Oct 2015. [Cited on pp. 11]

[159] Bangtian Liu, Chengyao Wen, Anand D. Sarwate, and Maryam Mehri Dehnavi. A unified optimization approach for sparse tensor operations on GPUs. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 47–57, Sept 2017. [Cited on pp. 160]

[160] Joseph W. H. Liu. Computational models and task scheduling for parallel sparse Cholesky factorization. *Parallel Computing*, 3(4):327–342, 1986. [Cited on pp. 131, 134]

[161] Joseph W. H. Liu. A graph partitioning algorithm by node separators. *ACM Transactions on Mathematical Software*, 15(3):198–219, 1989. [Cited on pp. 140]

[162] Joseph W. H. Liu. Reordering sparse matrices for parallel elimination. *Parallel Computing*, 11(1):73 – 91, 1989. [Cited on pp. 131]

[163] Liang Liu, Jun (Jim) Xu, and Lance Fortnow. Quantized BvND: A better solution for optical and hybrid switching in data center networks. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing*, pages 237–246, Dec 2018. [Cited on pp. 9]

[164] Zvi Lotker, Boaz Patt-Shamir, and Seth Pettie. Improved distributed approximate matching. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 129–136, Munich, Germany, 2008. ACM. [Cited on pp. 59]

[165] László Lovász. On determinants, matchings, and random algorithms. In *FCT*, pages 565–574, 1979. [Cited on pp. 76]

[166] László Lovász and Micheal D. Plummer. *Matching Theory*. Elsevier Science Publishers, Netherlands, 1986. [Cited on pp. 76]

[167] Anna Lubiw. Doubly lexical orderings of matrices. *SIAM Journal on Computing*, 16(5):854–879, 1987. [Cited on pp. 151, 152, 153]

[168] Yuchen Ma, Jiajia Li, Xiaolong Wu, Chenggang Yan, Jimeng Sun, and Richard Vuduc. Optimizing sparse tensor times matrix on GPUs. *Journal of Parallel and Distributed Computing*, 129:99–109, 2019. [Cited on pp. 160]

[169] Jakob Magun. Greedy matching algorithms, an experimental study. *Journal of Experimental Algorithmics*, 3:6, 1998. [Cited on pp. 6]

[170] Marvin Marcus and R. Ree. Diagonals of doubly stochastic matrices. *The Quarterly Journal of Mathematics*, 10(1):296–302, 1959. [Cited on pp. 9]

[171] Nick McKeown. The iSLIP scheduling algorithm for input-queued switches. *EEE/ACM Transactions on Networking*, 7(2):188–201, 1999. [Cited on pp. 6]

[172] Amram Meir and John W. Moon. The expected node-independence number of random trees. *Indagationes Mathematicae*, 76:335–341, 1973. [Cited on pp. 67]

[173] John Mellor-Crummey, David Whaley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, Jun 2001. [Cited on pp. 160]

[174] Silvio Micali and Vijay V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science*, pages 17–27, Syracuse, NY, USA, 1980. IEEE. [Cited on pp. 6, 76]

[175] Orlando Moreira, Merten Popp, and Christian Schulz. Graph partitioning with acyclicity constraints. In *16th International Symposium on Experimental Algorithms, SEA*, London, UK, 2017. [Cited on pp. 16, 17, 28, 29, 32]

[176] Orlando Moreira, Merten Popp, and Christian Schulz. Evolutionary multi-level acyclic graph partitioning. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO*, pages 332–339, Kyoto, Japan, 2018. ACM. [Cited on pp. 17, 25, 32]

[177] Marcin Mucha and Piotr Sankowski. Maximum matchings in planar graphs via Gaussian elimination. In S. Albers and T. Radzik, editors, *ESA 2004*, pages 532–543, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. [Cited on pp. 76]

[178] Marcin Mucha and Piotr Sankowski. Maximum matchings via Gaussian elimination. In *FOCS '04*, pages 248–255, Washington, DC, USA, 2004. IEEE Computer Society. [Cited on pp. 76]

[179] Huda Nassar, Georgios Kollias, Ananth Grama, and David F. Gleich. Low rank methods for multiple network alignment. *arXiv e-prints*, page arXiv:1809.08198, Sep 2018. [Cited on pp. 128]

[180] Jaroslav Nešetřil and Patrice Ossona de Mendez. Tree-depth, subgraph coloring and homomorphism bounds. *European Journal of Combinatorics*, 27(6):1022–1041, 2006. [Cited on pp. 132]

[181] M. Yusuf Özkaya, Anne Benoit, Bora Uçar, Julien Herrmann, and Ümit V. Çatalyürek. A scalable clustering-based task scheduler for homogeneous processors using DAG partitioning. In *33rd IEEE International Parallel and Distributed Processing Symposium*, pages 155–165, Rio de Janeiro, Brazil, May 2019. [Cited on pp. 32]

[182] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, December 1987. [Cited on pp. 152, 153]

[183] Chiristos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, (Corrected, unabridged reprint of Combinatorial Optimization: Algorithms and Complexity originally published by Prentice-Hall Inc., New Jersey, 1982), New York, 1998. [Cited on pp. 29]

[184] Panos M. Pardalos, Tianbing Qian, and Mauricio G. C. Resende. A greedy randomized adaptive search procedure for the feedback vertex set problem. *Journal of Combinatorial Optimization*, 2(4):399–412, 1998. [Cited on pp. 142]

[185] Md. Mostofa Ali Patwary, Rob H. Bisseling, and Fredrik Manne. Parallel greedy graph matching using an edge partitioning approach. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 45–54, New York, NY, USA, 2010. ACM. [Cited on pp. 59]

[186] François Pellegrini. *SCOTCH 5.1 User's Guide*. Laboratoire Bordelais de Recherche en Informatique (LaBRI), 2008. [Cited on pp. 4, 16, 23]

[187] Daniel M. Pelt and Rob H. Bisseling. A medium-grain method for fast 2D bipartitioning of sparse matrices. In *IPDPS2014, IEEE 28th International Parallel and Distributed Processing Symposium*, pages 529–539, Phoenix, Arizona, May 2014. [Cited on pp. 52]

[188] Ioakeim Perros, Evangelos E. Papalexakis, Fei Wang, Richard Vuduc, Elizabeth Searles, Michael Thompson, and Jimeng Sun. SPARTan: Scalable PARAFAC2 for large and sparse data. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 375–384, New York, NY, USA, 2017. ACM. [Cited on pp. 156]

[189] Anh-Huy Phan, Petr Tichavský, and Andrzej Cichocki. Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations. *IEEE Transactions on Signal Processing*, 61(19):4834–4846, Oct 2013. [Cited on pp. 52]

[190] Juan C. Pichel, Francisco F. Rivera, Marcos Fernández, and Aurelio Rodríguez. Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs. *Microprocessors and Microsystems*, 36(2):65–77, 2012. [Cited on pp. 160]

[191] Matthias Poloczek and Mario Szegedy. Randomized greedy algorithms for the maximum matching problem with new analysis. In *IEEE 53rd Annual Sym. on Foundations of Computer Science (FOCS)*, pages 708–717, New Brunswick, NJ, USA, 2012. [Cited on pp. 58]

[192] Alex Pothen. *Sparse null bases and marriage theorems*. PhD thesis, Dept. Computer Science, Cornell Univ., Ithaca, New York, 1984. [Cited on pp. 68]

[193] Alex Pothen. The complexity of optimal elimination trees. Technical Report CS-88-13, Pennsylvania State Univ., 1988. [Cited on pp. 131]

[194] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16(4):303–324, 1990. [Cited on pp. 58, 68, 115]

[195] Alex Pothen, S. M. Ferdous, and Fredrik Manne. Approximation algorithms in combinatorial scientific computing. *Acta Numerica*, 28:541—633, 2019. [Cited on pp. 58]

[196] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://web.cse.ohio-state.edu/˜pouchet/software/polybench/*, 2012. [Cited on pp. 25, 26]

[197] Michael O. Rabin and Vijay V. Vazirani. Maximum matchings in general graphs through randomization. *Journal of Algorithms*, 10(4):557–567, December 1989. [Cited on pp. 76]

[198] Daniel Ruiz. A scaling algorithm to equilibrate both row and column norms in matrices. Technical Report TR-2001-034, RAL, 2001. [Cited on pp. 55, 56]

[199] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms – ESA 2011: 19th Annual European Symposium, September 5-9, 2011*, pages 469–480, Saarbrücken, Germany, 2011. [Cited on pp. 4, 16, 23]

[200] Robert Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8(3):256–276, 1982. [Cited on pp. 131]

[201] R. Oguz Selvitopi, Muhammet Mustafa Ozdal, and Cevdet Aykanat. A novel method for scaling iterative solvers: Avoiding latency overhead of parallel sparse-matrix vector multiplies. *IEEE Transactions on Parallel and Distributed Systems*, 26(3):632–645, 2015. [Cited on pp. 51]

[202] Richard Sinkhorn and Paul Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21:343–348, 1967. [Cited on pp. 55, 69, 114]

[203] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools. `http://frostt.io/`, 2017. [Cited on pp. 52, 126, 156, 161]

[204] Shaden Smith and George Karypis. A medium-grained algorithm for sparse tensor factorization. In *2016 IEEE International Parallel and*

Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016, pages 902–911, 2016. [Cited on pp. 37, 52]

[205] Shaden Smith and George Karypis. SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit (Version 1.1.1). Available from `https://github.com/ShadenSmith/splatt`, 2016. [Cited on pp. 37, 52, 157]

[206] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *29th IEEE International Parallel & Distributed Processing Symposium*, pages 61–70, Hyderabad, India, May 2015. [Cited on pp. 11, 36, 37, 38, 39, 42, 158, 159]

[207] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984. [Cited on pp. 149]

[208] Bora Uçar and Cevdet Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, 2004. [Cited on pp. 47, 51]

[209] Bora Uçar and Cevdet Aykanat. Revisiting hypergraph models for sparse matrix partitioning. *SIAM Review*, 49(4):595–603, 2007. [Cited on pp. 42]

[210] Vijay V. Vazirani. An improved definition of blossoms and a simpler proof of the MV matching algorithm. *CoRR*, abs/1210.4594, 2012. [Cited on pp. 76]

[211] David W. Walkup. Matchings in random regular bipartite digraphs. *Discrete Mathematics*, 31(1):59–64, 1980. [Cited on pp. 59, 67, 76, 81, 114, 121]

[212] Chris Walshaw. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research*, 131(1):325–372, Oct 2004. [Cited on pp. 25]

[213] Eric S. H. Wong, Evangeline F. Y. Young, and Wai-Kei Mak. Clustering based acyclic multi-way partitioning. In *Proceedings of the 13th ACM Great Lakes Symposium on VLSI*, GLSVLSI '03, pages 203–206, New York, NY, USA, 2003. ACM. [Cited on pp. 4]

[214] Tao Yang and Apostolos Gerasoulis. DSC: scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994. [Cited on pp. 131]

[215] Albert-Jan Nicholas Yzelman and Dirk Roose. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):116–125, Jan 2014. [Cited on pp. 160]