HAL
open science

On matchings and related problems in graphs, hypergraphs, and doubly stochastic matrices

Ioannis Panagiotas

To cite this version:

Ioannis Panagiotas. On matchings and related problems in graphs, hypergraphs, and doubly stochastic matrices. Data Structures and Algorithms [cs.DS]. Université de Lyon, 2020. English. NNT : 2020LYSEN068 . tel-03011794

HAL Id: tel-03011794

https://theses.hal.science/tel-03011794v1

Submitted on 18 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Numéro National de Thèse : 2020LYSEN068

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée par

l'École Normale Supérieure de Lyon

École Doctorale N°512

École Doctorale en Informatique et Mathématiques de Lyon

Spécialité : Informatique

présentée et soutenue publiquement le 09/10/2020, par :

Ioannis PANAGIOTAS

On matchings and related problems in graphs, hypergraphs, and doubly stochastic matrices

Sur les couplages et les problèmes liés dans les graphes, les hypergraphes et les matrices doublement stochastiques

Devant le jury composé de :

Clémence	MAGNIEN	Directrice de recherche, CNRS	*Rapporteure*
Alex	POTHEN	Professeur, Purdue Univ., Etats-Unis	*Rapporteur*
Marthe	BONAMY	Chargée de recherche, CNRS	*Examinatrice*
Dimitrios	THILIKOS	Directeur de recherche, CNRS	*Examinateur*
Bora	UÇAR	Chargé de recherche, CNRS	*Directeur de thèse*
Fanny	DUFOSSE	Chargée de recherche, INRIA	*Co-Encadrante de thèse*

Contents

Acknowledgements . . . v
Résumé français . . . vi

1 Introduction **1**
1.1 Undirected graphs . . . 1
1.2 Hypergraphs . . . 3
1.3 Sparse matrices . . . 4
1.3.1 The permanent function . . . 4
1.3.2 Doubly stochastic matrices . . . 5
1.4 Sparse tensors . . . 6
1.5 Structure of the thesis . . . 7

2 Matchings in bipartite graphs **9**
2.1 A survey on matching heuristics . . . 10
2.2 An examination of the Karp–Sipser algorithm . . . 11
2.2.1 An expected $\mathcal{O}(m \log n)$-time algorithm . . . 12
2.2.2 An implementation with list caching . . . 13
2.2.3 An alternating component approach . . . 14
2.2.4 Fast recovery of the matching . . . 16
2.2.5 Experiments . . . 17
2.2.6 Related work . . . 25
2.3 Scaling based near-optimal randomized algorithms . . . 25
2.3.1 2OUTMC: Monte Carlo on 2-out graphs . . . 26
2.3.2 TRUNCRW: Truncated random walk with nonuniform sampling . . . 31
2.3.3 Experiments . . . 34
2.4 A scaling based derandomized algorithm . . . 43
2.4.1 The derandomization . . . 45
2.4.2 Some preliminary experiments . . . 48
2.5 Concluding remarks . . . 49

3 Matchings in undirected graphs **51**
3.1 ONE-OUT: The main heuristic . . . 51
3.1.1 ONE-OUT: Analysis . . . 53
3.1.2 Two variants of ONE-OUT . . . 58
3.2 Experiments . . . 59
3.2.1 On real-life graphs . . . 59
3.2.2 On a hard synthetic instance for KS_{R1} . . . 62
3.2.3 On large-scale graphs . . . 63

3.3 Concluding remarks . . . 63

4 Matchings in hypergraphs **65**
4.1 Heuristics for maximum d-dimensional matching . . . 66
4.1.1 A Greedy heuristic for Max-d-DM . . . 66
4.1.2 KarpSipserH for Max-d-DM . . . 66
4.1.3 KarpSipserHScaling for Max-d-DM . . . 67
4.1.4 Hypergraph matching via pseudo scaling . . . 69
4.1.5 Reduction to bipartite graph matching . . . 69
4.1.6 Performing local search . . . 71
4.2 Experiments . . . 71
4.2.1 On random hypergraphs . . . 71
4.2.2 On synthetic hypergraphs . . . 74
4.2.3 On real-life hypergraphs . . . 76
4.2.4 Comparison with an independent set solver . . . 76
4.3 Concluding remarks . . . 77

5 Counting the number of perfect matchings in graphs **79**
5.1 Theoretical background . . . 80
5.2 Related work . . . 80
5.3 The proposed algorithm and its analysis . . . 81
5.3.1 The algorithm . . . 82
5.3.2 The analysis . . . 83
5.4 An estimator for undirected graphs . . . 87
5.4.1 The algorithm and its analysis . . . 87
5.4.2 Filtering out redundant edges . . . 90
5.5 Experiments . . . 91
5.5.1 On bipartite graphs . . . 91
5.5.2 On general, undirected graphs . . . 96
5.6 Concluding remarks . . . 98

6 Results on the Birkhoff–von Neumann decomposition **101**
6.1 The two heuristics . . . 102
6.2 Analysis of the two heuristics for computing BvN decompositions . . . 102
6.3 Heuristics for $\textsc{Greedy}_{\text{BVN}}$ . . . 106
6.4 Experiments on real-life matrices . . . 107
6.5 Concluding remarks . . . 109

7 Conclusion **111**
7.1 Summary of the chapters . . . 111
7.2 Future work . . . 112

List of publications **121**

List of Algorithms

2.1 2OUTMC: Monte Carlo on 2-out graphs 27
2.2 SAMPLE: Algorithm to sample a random neighbor of a column vertex c with d_c neighbors 33
2.3 ONESIDED: Matching heuristic 44
2.4 ONESIDEDDERAND: The derandomized variant of ONESIDED 47

3.1 ONE-OUT: Heuristic for matching in undirected graphs 52
3.2 KARPSIPSER$_{\text{ONE-OUT}}$: Specialized Karp–Sipser for 1-out graphs 54

4.1 KARPSIPSERHSCALING: The scaling-based extension of Karp–Sipser in hypergraphs 68

5.1 ESTSCALINGPERM: Permanent estimation 83
5.2 ESTSCALINGMTC: Estimation of the number of perfect matchings in graphs . . . 88

6.1 GENERALIZED-BIRKHOFF: Template to find a BvN decomposition 102

Acknowledgments

I would like first and foremost to thank my supervisor Bora Uçar for our collaboration during the last three and a half years. He was always very positive, approachable and supportive to me. He was always more than keen to discuss with me when I was facing some issues. His comments were always helpful and I learned a lot through our conversations about research and writing papers. His enthusiasm for the matching problem is probably unparalleled!

I am also thankful for having Fanny Dufossé as my second supervisor. Our frequent visits to each other during the first year, where Bora was away in the USA, really helped me stay focused. I was also very lucky to work with Kamer Kaya from Sabanci University in Turkey who essentially became my unofficial third supervisor. I was really amazed by Kamer's ability to tidy up a paper quickly and efficiently. I would also like to thank Johannes Langguth as well as all the people from the Simula Laboratory for hosting me in Oslo for a week. It was a very rewarding experience for me to work in a different environment. I really hope to have the chance to collaborate with Bora, Fanny, Kamer, and Johannes in the future.

I am proud to have been part of ROMA. The climate in the team is very welcoming and friendly. I very much enjoyed the lighthearted conversations we were having with the other PhD students in the team. They were always a nice break from the troubles of work. Hopefully we can all meet again in the future (and perhaps finally do a hike together). I am also thankful to Marie Bozo, Evelyne Blesle, Laetitia Lecot, Solene Audoux and Virginie Bouyer for helping me with the various administration tasks.

I would like to thank Clémence Magnien and Alex Pothen for agreeing to review the manuscript of my thesis in detail and act as reporters in my defense. I am also grateful to Marthe Bonamy and Dimitrios Thilikos for agreeing to act as examinators in my defense.

I have nothing but the biggest gratitude for my family and my sister, Katerina, in particular. I will never forget their support and how much they stood up for me. Knowing that I have so many people caring for me gave me all the strength I needed. To that end, I would like to dedicate this thesis in the memory of my parents and especially of my father. He supported me emotionally and financialy throughout all my studies and unfortunately he passed away before he could see me graduate.

I would also like to thank all my friends both here in Lyon as well as those in Greece or other places. For those I met in Lyon, I am thankful for all the drinks, parties, board games and the hikes we enjoyed together. To the rest, thank you for keeping in touch with me during this time. In fact, I would like to give an extra thanks to a couple of friends with whom I communicated daily.

Finally, I feel I have to thank God for surrounding me with good and caring people as well as for all things -good or bad- that have happened to me.

Résumé français

Cette thèse examine quatre problèmes différents et connexes qui se posent dans le domaine du calcul scientifique combinatoire (CSC). Le lien de connexion entre les quatre problèmes examinés est le problème fondamental des couplages, qui cherche le plus grand ensemble d'arêtes disjointes dans un graphe ou un hypergraphe. Ces problèmes sont ceux du couplage de cardinalité maximale dans les graphes et dans les hypergraphes, l'estimation du nombre de couplages parfaits, et la décomposition de Birkhoff–von Neumann des matrices bistochastiques. L'étude de ces problèmes est motivée par leur utilité dans plusieurs domaines d'application.

La recherche en CSC s'articule autour de la formulation d'un problème de calcul scientifique à l'aide d'un modèle combinatoire et de l'étude des problèmes algorithmiques sous-jacents associés à ce modèle. La recherche en CSC est donc un mélange de travail théorique et pratique.

Nous examinons les quatre problèmes à la fois théoriquement et expérimentalement. L'accent mis sur la théorie nous permet de discuter, d'analyser et de prouver les propriétés des algorithmes examinés, tandis que le côté expérimental de la thèse démontre les améliorations et les avantages possibles de l'utilisation des algorithmes et des solutions proposés.

Le chapitre 2 examine le problème de couplage dans les graphes bipartis et se compose de trois parties. Dans la première partie, nous considérons, une heuristique de couplage bien connue. Nous décrivons une mise en œuvre efficace et examinons sa complexité dans le pire cas. Nous comparons expérimentalement sa variante plus simple largement utilisée et montrons des cas pour lesquels l'algorithme complet donne de meilleures performances. Dans la deuxième partie du chapitre, nous examinons deux algorithmes de couplage probabilistes exacts pour deux classes spéciales de graphes bipartis qui sont les graphes aléatoires de type 2-out et les graphes réguliers de degré d. Nous généralisons ces deux algorithmes et les transformons en heuristiques pratiques pour des graphes bipartis arbitraires. L'élément clé de ces généralisations est la mise à l'échelle de la matrice, que nous appliquons sur la matrice d'adjacence d'un graphe donné. Les résultats expérimentaux montrent que les heuristiques sont rapides et permettent d'obtenir des couplages quasi optimales. Elles sont également plus robustes que les heuristiques de pointe, et sont généralement plus utiles comme routines d'initialisation. La troisième partie du chapitre consiste en un bref résultat théorique sur la dérandomisation d'une heuristique randomisée connue. Nous proposons un moyen de remplacer les décisions aléatoires de l'algorithme randomisé par des décisions déterministes sans réduire la garantie d'approximation.

Le chapitre 3 examine le problème de couplage maximal dans les graphes non dirigés en général. Comme dans le chapitre précédent, nous discutons des algorithmes efficaces qui peuvent trouver de bonnes approximations des couplages assez rapidement pour l'initialisation. Ce chapitre s'appuie sur les deux dernières parties du chapitre 2 en considérant les algorithmes basés sur la mise à l'échelle de la matrice. Plus précisément, les algorithmes examinés sont basés sur la théorie connue des graphes bipartis de type 1-out. Nous étendons la théorie pour qu'elle s'applique aux graphes généraux non orientés. Nous montrons que notre heuristique principale

a une garantie d'approximation d'environ $0,866 - \log(n)/n$ pour un graphe avec n sommets. Nous fournissons plusieurs expériences qui vérifient les résultats théoriques.

Le chapitre 4 examine le problème de la recherche d'un couplage de cardinalité maximale dans un hypergraphe d-partie, d-uniforme. Comme le problème est NP-dur, nous concevons plusieurs heuristiques. Certaines de nos heuristiques sont des généralisations d'heuristiques connues pour le problème de couplages de cardinalité maximale dans les graphes. Nous proposons également une nouvelle heuristique basée sur la mise à l'échelle du tenseur. Cette heuristique s'inspire des propriétés de la mise à l'échelle matricielle pour les graphes bipartis du chapitre 2. Les expériences sur les hypergraphes aléatoires, synthétiques et réels montrent que cette nouvelle heuristique est mieux que les autres.

Le chapitre 5 étudie des méthodes randomisées efficaces pour compter approximativement le nombre de couplages parfaits dans les graphes bipartites et les graphes généraux non dirigés. L'approche examinée utilise la mise à l'échelle de la matrice pour échantillonner un couplage parfait du graphe et utilise la probabilité de sélection pour retourner une approximation du nombre de couplages parfaites dans le graphe. Cette approche présente des similitudes avec l'approche basée sur la mise à l'échelle du chapitre 4 dans le sens où la mise à l'échelle est appliquée à chaque étape pour modéliser l'état actuel du graphe aussi précisément que possible. L'analyse expérimentale sur des graphes aléatoires et réels montre des améliorations dans l'approximation par rapport aux méthodes connues et similaires de la littérature.

Le chapitre 6 examine le problème Min-BvN-Decomp défini pour les matrices bistochastiques. Nous considérons deux heuristiques connues appelées $\text{Greedy}_{\text{BVN}}$ et Birkhoff pour ce problème. Nous montrons les limites de performance pour ces deux heuristiques. Nos observations ainsi que les travaux précédents nous permettent de mieux expliquer pourquoi $\text{Greedy}_{\text{BVN}}$ a une performance supérieure à Birkhoff. Nous considérons ensuite deux modifications de $\text{Greedy}_{\text{BVN}}$ et démontrons expérimentalement que ces modifications peuvent conduire à une décomposition de BvN avec moins de matrices.

Le chapitre 7 conclut la thèse. Nous fournissons un résumé avec les points les plus importants couverts dans chaque chapitre et nous discutons également des travaux futurs potentiels.

Chapter 1

Introduction

This thesis investigates four different and related problems that arise in the area of *Combinatorial Scientific Computing (CSC)*. The connecting link between the four examined problems is the fundamental problem of matching [86], which asks for the largest set of disjoint edges in a graph or hypergraph. These problems are that of maximum cardinality matching in graphs and hypergraphs, the estimation of the number of perfect matchings in graphs, and the Birkhoff–von Neumann decomposition of doubly stochastic matrices. The study of these problems is motivated by their usefulness in several domains and applications. We will mention some of these applications in this chapter.

Research in CSC revolves around the formulation of a scientific computing problem using a combinatorial model and the examination of the underlying algorithmic problems associated with this model. Research in CSC is thus a mixture of theoretical and practical work.

We examine the four problems both theoretically as well as experimentally. The focus in theory allows us to discuss, analyze, and prove properties of the examined algorithms, while the experimental side of the thesis demonstrates the possible improvements and benefits of using the proposed algorithms and solutions.

The rest of the chapter is organized as follows. In Sections 1.1–1.4 we provide a general overview of the research area and introduce all the necessary background and material which are needed to comprehend the results of the thesis. Section 1.5 then summarizes the remaining chapters and gives a brief overview of the obtained results.

1.1 Undirected graphs

An *undirected graph* $G = (V, E)$ consists of a set of vertices V and a set of edges E of the form $e = (u, v)$, where $u, v \in V$. The vertices u and v are called the endpoints of e and are considered as neighbors of each other. The term *degree* of a vertex refers to the number of its neighbors. A *degree-k* vertex is a vertex with degree exactly k. Degree-1 and degree-2 vertices are especially important in the context of this thesis as it will be discussed in Chapter 2. A graph is *complete* when there exists an edge between any two vertices of V and is represented with K_n, where $n = |V|$. An *independent set* is a set of vertices such that there does not exist an edge between any two vertices from this set.

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, 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 cycle is

simple if there are no other vertex repetitions. A *tree* is a connected graph with no cycles. The number of edges in a path, represented with ℓ, is referred to as the *length* of the path. A path is *even* or *odd* depending on the parity of ℓ.

Graphs and graph-based models have found uses in many areas both in the industrial and the academic world. Among others, they can be used to naturally model road networks, social networks, and molecules. In the field of Combinatorial Scientific Computing graphs are commonly used in sparse linear solvers and sparse matrix computations in general [30, 34]. Additional uses of graphs in CSC, for example on *automatic differentiation* are discussed in related textbooks [94].

Definition 1.1. *A* matching $\mathcal{M}$ *in a graph* $G = (V, E)$ *is a subset of disjoint edges of* E *such that no two edges in* $\mathcal{M}$ *share a common vertex.*

Given a matching $\mathcal{M}$, we call a vertex u *matched*, if there exists an edge of the form (u, v) in $\mathcal{M}$. In which case v is called u's *mate* and vice versa. Note that by the definition of matching, a vertex can have at most one mate. If vertex u does not have a mate in $\mathcal{M}$, then u is called *unmatched* or *free*. The *cardinality* of $\mathcal{M}$, represented with $|\mathcal{M}|$, corresponds to the number of edges in $\mathcal{M}$. A matching $\mathcal{M}$ is *maximal* if it cannot be extended with new edges. In this case, all edges of E which are not included in $\mathcal{M}$ have at least one endpoint in the vertices of $\mathcal{M}$. A matching $\mathcal{M}$ with the largest cardinality is called a *maximum cardinality matching*. We will use $\mathcal{M}^*$ to refer the maximum cardinality matching of a given graph G. If $\mathcal{M}^*$ matches all vertices of V, then it is additionally called *perfect*. An *augmenting path* with respect to a matching $\mathcal{M}$ is a path which starts with a free vertex and ends at another free vertex such that every second edge is in $\mathcal{M}$. A matching is maximum if and only if there are no augmenting paths [10].

Finding a maximum cardinality matching has been one of the fundamental problems in computer science. Its applications include assignment [18] and scheduling [89] problems among others. The lowest worst-case time complexity of the known algorithms is $\mathcal{O}(m\sqrt{n})$ for a graph with n vertices and m edges [14, 51, 91]. This complexity can be prohibiting for large instances. For this reason there is significant interest in algorithms which can find large matchings in linear or near linear time [98]. These large matchings can then be given as initialization to an exact algorithm in order to speed-up the process of finding a maximum cardinality matching. The practical uses of such initialization techniques [85] are well known and demonstrated. Additionally there exist applications [89] where large approximate matchings suffice by themselves, i.e., we do not strictly need a maximum one.

An undirected graph $G = (V, E)$ is *bipartite* if and only if V can be partitioned into two disjoint sets V_R and V_C such that $V_R \cup V_C = V$, and no vertices from the same partition are neighbors. Bipartite graphs do not contain odd cycles. We use K_{n_r,n_c} to represent the complete bipartite graph with $|V_R| = n_r$ and $|V_C| = n_c$. The lack of odd cycles makes finding the problem of maximum cardinality matching simpler on bipartite graphs. The best known algorithms [62] run in $\mathcal{O}(m\sqrt{n})$ time for a graph with n vertices per side and m edges. While as seen, the best known complexity is asymptotically the same for both general undirected graphs and bipartite graphs, algorithms for the bipartite case are less complex, and consequently have better practical performance.

A k-out subgraph G_k of a host graph G is defined by allowing each vertex in G to randomly select uniformly k of its neighbors, and the union of all selections forms the edge set of G_k. Walkup [107] showed that a random 1-out graph in the pure random bipartite k-out setting (where the host graph is the complete bipartite graph) almost surely does not have a perfect matching. He further showed that when $k \geq 2$ the resulting G_k has a perfect matching with high

probability, again focusing exclusively on complete bipartite graphs. Karoński and Pittel [69], later with Overman [68], continued to examine complete bipartite graphs, and focused on two models in between 1-out and 2-out bipartite graphs. Both models start by generating a random 1-out bipartite subgraph. The two models differ in their treatment of subsequent steps. In the first model, the vertices that were not selected at all choose one more neighbor. In the second model, the vertices that were selected at most once choose one more neighbor. It was initially claimed [69] that subgraphs generated by the first model had a perfect matching with high probability. This claim was shown to be false in the same paper that proved the existence of a perfect matching for the subgraphs of the second model [68].

Frieze [46] generalized the result of Walkup and showed that k-out graphs of complete undirected graphs also have a perfect matching with high probability for $k \geq 2$. The problem of Hamiltonian paths [15] and k-connectivity [43] have also been studied in the context of subgraphs of complete graphs generated with the k-out model.

On the other hand, as far as we are aware, there is a scarcity of results about k-out graphs sampled from an arbitrary host graph. Frieze and Johansson [45] investigated some other properties of G_ks on host graphs where the minimum degree of a vertex is at least $n/2$. Ghaffari et al. [52] and Holm et al. [61] examined applications of k-out graphs in connectivity problems. Dufossé et al. [38] proposed to base the random decisions on matrix scaling and showed that 1-out subgraphs of graphs generated using the aforementioned method have a maximum cardinality matching of around 0.866 of the maximum one in the host graph under some conditions. We will investigate the k-out model and its applications for matchings in arbitrary host graphs in Chapters 2 and 3.

1.2 Hypergraphs

A hypergraph $H = (V, E)$ consists of a finite set V and a collection E of subsets of V. The set V is called the set of vertices, and the collection E is called the set of hyperedges. A hypergraph 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. When $d = 2$ the corresponding hypergraph is a bipartite graph. For a higher d, a d-partite, d-uniform hypergraph is an extension of bipartite graphs to d dimensions.

A lot of problems that appear in graphs can be generalized to hypergraphs. A matching in a hypergraph is defined as a set of hyperedges such that no two hyperedges share a common vertex. Definitions such as *maximal matching* or *perfect matching* are defined in accordance with their equivalents on graphs.

We refer to Max-d-DM as the problem of finding a maximum cardinality matching on a d-partite, d-uniform hypergraph. Max-d-DM for $d \geq 3$ is NP-Complete; the 3-partite case in particular was part of Karp's 21 NP-Complete problems [70]. Max-d-DM has been studied mostly in the context of local search algorithms [64], and the best known algorithm is due to Cygan [27] who proposed an $((d + 1 + \varepsilon)/3)$-approximation, building on previous work [29, 58]. It is NP-Hard to approximate Max-3-DM within 98/97 [11]. 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 [59]. Note that Max-d-DM is a special case of the d-Set-Packing problem [60]. Hence, results such as d-Set-Packing being hard to approximate within a factor of $\mathcal{O}(d/\log d)$ [60] are also relevant for Max-d-DM.

The maximum/perfect set packing problem has many applications, including combinatorial auctions [56] and personnel scheduling [47]. Matchings in hypergraphs can also be used in the

coarsening phase of multilevel hypergraph partitioning tools [20], when the input is d-uniform and d-partite, such as those used in modeling and partitioning tensors [77].

The *k-out random hypergraph model* generalizes the random k-out graph model that was introduced earlier in the chapter. Given a hypergraph $H = (V, E)$, each vertex $u \in V$ selects k hyperedges from the set $E_u = \{e : e \in E, u \in e\}$ in a uniformly random fashion and the union of these edges forms the hyperedge set of the subhypergraph. For the d-partite, d-uniform case in particular we have $E_u = \{e : |e \cap V_i| = 1 \text{ for } 1 \leq i \leq d, u \in e, e \in E\}$. Hence (ignoring the duplicate ones), such hypergraphs have around $d \times k \times n$ hyperedges. Devlin and Kahn [32] investigate fractional matchings in these hypergraphs, and mention in passing that k should be exponential in d to ensure that a perfect matching exists when the host hypergraph is complete, i.e., contains all possible hyperedges.

1.3 Sparse matrices

We use bold upper case letters to represent matrices, as in $\mathbf{A}$. The entries in the matrix are shown with lower-case letters and subscripts, e.g., $a_{i,j}$ denotes the entry of the matrix $\mathbf{A}$ at the ith row and jth column. The column ids of the nonzeros in the ith row of $\mathbf{A}$ are represented as $\mathbf{A}(i, :)$. With $\mathbf{A}_{ij}$ we denote the submatrix of matrix $\mathbf{A}$ obtained by deleting the ith row and the jth column. A permutation matrix is a square matrix such that each row and each column contain exactly one nonzero value equal to 1. If an $n \times n$ permutation matrix has all its nonzeros along the main diagonal (i.e., $a_{ij} = 1$ iff $i = j$), it is called the *identity matrix* and symbolized with $\mathbf{I}_n$.

In the chapters to follow, we will make extensive use of matrices, which we store in the following memory efficient way. Under the *Compressed Sparse Column (CSC)* representation, only the nonzero entries of $\mathbf{A}$ are kept in memory. The CSC format stores information about the nonzero pattern of $\mathbf{A}$ in two one-dimensional arrays `IDS` and `XIDS`. All neighbors of column c are stored in consecutive positions `XIDS[`c`]`,..., `XIDS[`$c+1$`]`-1 of the array `IDS`. An optional one-dimensional array `VALUE` can be used to store the values of $\mathbf{A}$, such that `VALUE[i]` stores the value of the element stored in `IDS[i]`. The *Compressed Sparse Row (CSR)* format keeps similar information for the rows and is defined accordingly.

Any graph $G = (V, E)$ with $|V| = n$ can be represented by symmetric $n \times n$ matrix $\mathbf{A}_G$, which is called the *adjacency matrix* of G. We will use $\mathbf{A}$ instead of $\mathbf{A}_G$ when the context is clear. In this matrix, we have $a_{ij} = a_{ji} = 1$ if and only if the edge (u_i, u_j) exists. Here, we associate a distinct index in the matrix for each of the vertices in V. If the graph $G = (V, E)$ is bipartite, with $|V_R| = n_r$ and $|V_C| = n_c$, we can instead store G in an $n_r \times n_c$ matrix $\mathbf{A}_G$ with less memory overhead. We associate for each vertex of V_R a distinct row, and for each vertex of V_C a distinct column. Then, the entry $a_{ij} = 1$ if and only if the edge (r_i, c_j) exists. Note that unlike the above representation, $\mathbf{A}_G$ in here is not necessarily symmetric and edges in the main diagonal are allowed.

A matrix can be represented as a bipartite graph in a similar way. The edge (r_i, c_j) exists in E if and only if the value of a_{ij} is nonzero. A perfect matching in a bipartite graph is thus equivalent to a permutation of its equivalent adjacency matrix and vice versa.

1.3.1 The permanent function

The permanent of an $n \times n$ square matrix $\mathbf{A}$ is defined as $Per(\mathbf{A})$ is equal to $\sum_\sigma \prod_i a_{i,\sigma(i)}$, where the summation runs over all permutations σ of $1, \ldots, n$. The value of the permanent of

the matrix representation $\mathbf{A}_G$ of a bipartite graph G is equal to the number of perfect matchings in G. We can similarly use the adjacency matrix representation to calculate the number of perfect matchings in undirected graphs. Note that however this quantity can be different than the value of the permanent in the corresponding adjacency matrix.

Approximating the permanent is a well-studied problem. Valiant [106] showed the problem to be #P-Complete. Jerrum et al. [66] discussed an approach using Markov Chains which can provide an $(1+\varepsilon)$-approximation for the permanent in fully polynomial time, with $\tilde{O}(n^{10})$ complexity. Their Markov Chain Monte Carlo (MCMC) approach makes use of the underlying graph being bipartite, and their techniques cannot be generalized easily to the general graph case. Rasmussen [99] proposed a different approach based on employing the mean of several unbiased estimators. We will examine this approach in detail in Chapter 5.

Aside from the theoretical importance of the permanent, there exist also practical applications in the field of statistical mechanics, namely in the monomer-dimer model [63]. Under this model, a set of points in a lattice is covered by a non-overlapping arrangement of monomers (molecules which occupy a single point) and dimers (molecules which occupy two neighboring points). All possible monomer-dimer coverings (i.e., placings of monomers and dimers to fully cover all points) for a given lattice define the *configuration space*. Given a lattice, we are interested in calculating the cardinality of the configuration space. A lattice can be represented as a bipartite graph G where each vertex represents a point and two vertices are neighbors if their corresponding points are adjacent in the lattice. Now the problem of calculating the cardinality of the configuration space is equivalent to calculating the number of matchings in the graph G. To see why, let M be matching in G. One can then visualize the matched edges of M as dimers, and all free vertices as monomers, which will then fully cover the lattice. The number of all matchings in G can be calculated from the permanent of the following matrix $\mathbf{B} = \begin{pmatrix} \mathbf{A}_G & \mathbf{I}_{n\times n} \\ \mathbf{1}_{n\times n} & \mathbf{1}_{n\times n} \end{pmatrix}$, where $\mathbf{1}_{n\times n}$ the $n \times n$ matrix with all values equal to 1, as $\dfrac{Per(\mathbf{B})}{n!}$ [63].

1.3.2 Doubly stochastic matrices

Here, we introduce one of the most essential concepts of the thesis, which is the class of doubly stochastic matrices, as well as the notion of matrix scaling.

Definition 1.2. *An $n \times n$ matrix $\mathbf{A} \neq 0$ is called* doubly stochastic *if every entry $a_{ij} \geq 0$ for all $i, j \leq n$, and in addition the sum of entries in each row and in each column is equal to 1.*

Our results make extensive use of the fact that a row or a column of a doubly stochastic matrix can be considered as a probability distribution.

A well-known theorem due to Birkhoff [13] states that for a given doubly stochastic matrix $\mathbf{A}$ there exist $\alpha_1, \alpha_2, \ldots, \alpha_k \in (0, 1]$ with $\sum_{i=1}^{k} \alpha_i = 1$ and k different 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 the Birkhoff–von Neumann (BvN) decomposition. Such decompositions are not unique and in general a doubly stochastic matrix can have several valid BvN decompositions. The problem Min-BvN-Decomp, which asks for the decomposition with the smallest k, is strongly NP-Complete [39].

Marcus and Ree [88] showed that a dense matrix can be decomposed with $k \leq n^2 - 2n + 2$ permutation matrices. A tighter upper bound of $k \leq \tau - 2n + 2$ can be guaranteed for sparse,

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \qquad \mathbf{S} = \begin{pmatrix} 0.618 & 0.382 & 0 \\ 0 & 0.382 & 0.618 \\ 0.382 & 0.236 & 0.382 \end{pmatrix}$$

Figure 1.1 – A nonnegative matrix $\mathbf{A}$ with total support and its corresponding scaled matrix $\mathbf{S} = \mathbf{RAC}$.

fully indecomposable matrices with τ nonzeros [16, 17]. Brualdi [16] gave lower bounds on the number of permutation matrices in any BvN decomposition of a given matrix.

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 [22]. 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 variation of this problem appears in routing in data centers [83]. BvN decompositions are also used to build preconditioners for solving sparse linear systems [9]. Here, the number k of the permutation matrices is related to the cost of applying the preconditioner.

A matrix is said to have *support* if there is a perfect matching in the associated bipartite graph. Furthermore, if each nonzero can be put in at least one perfect matching, then the matrix is said to have *total support*. By Birkhoff's theorem, a doubly stochastic matrix necessary has total support.

Any nonnegative matrix $\mathbf{A}$ with total support can be scaled with two unique positive diagonal matrices $\mathbf{R}$ and $\mathbf{C}$ such that the matrix $\mathbf{S} = \mathbf{RAC}$ is doubly stochastic. 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 [101]. Figure 1.1 shows the $\mathbf{S}$ matrix of a given 3×3 matrix $\mathbf{A}$. The Sinkhorn–Knopp algorithm [101] 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. If $\mathbf{A}$ is symmetric and scalable, then $\mathbf{S} = \mathbf{RAR}$ is doubly stochastic. While the Sinkhorn–Knopp algorithm obtains this symmetric, doubly stochastic matrix in the limit, there are other iterative algorithms that maintain symmetry all along the way [79, 80]. Idel [65] gives a comprehensive survey of known results for computing doubly stochastic scalings. Recently, a tighter analysis of the Sinkhorn–Knopp algorithm [21] has been carried out, and other efficient algorithms based on convex optimization have been proposed [3, 25].

1.4 Sparse tensors

Tensors are multidimensional arrays, generalizing matrices to higher orders. We will use similar notation with matrices to refer and describe tensors. Let $\mathbf{T}$ be a d-dimensional tensor whose size is $n_1 \times \cdots \times n_d$. The elements of $\mathbf{T}$ are shown with $\mathbf{T}_{i_1,\ldots,i_d}$, where $i_j \in \{1, \ldots, n_j\}$. A marginal is a $(d-1)$-dimensional section of a d-dimensional tensor, obtained by fixing one of its indices.

Similar to how matrices can be used to represent graphs, one can use a tensor to represent a given d-partite, d-uniform hypergraph. This is done by associating each tensor dimension with a vertex class. Let $|V_i| = n_i$, and the tensor $\mathbf{T} \in \{0,1\}^{n_1 \times \cdots \times n_d}$ have a nonzero element $\mathbf{T}_{v_1,\ldots,v_d}$ iff $(v_1, \ldots, v_d)$ is a hyperedge of H. Then, $\mathbf{T}$ is called the *adjacency tensor* of H. A tensor where

each marginal contains exactly one nonzero entry (equal to one) is called a permutation tensor. As in the bipartite case, there is a correspondence between perfect matchings in a hypergraph and permutation tensors in the adjacency tensor representation of the hypergraph.

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. A d-stochastic tensor where each marginal contains exactly one nonzero entry (equal to one) is called a permutation tensor. Franklin and Lorenz [44] showed that if a nonnegative tensor $\mathbf{T}$ has the same zero-pattern as a d-stochastic tensor $\mathbf{B}$, then one can find a set of d vectors $x^{(1)}, x^{(2)}, \ldots, x^{(d)}$ such that $\mathbf{T}_{i_1,\ldots,i_d} \cdot x^{(1)}_{i_1} \cdot \cdots \cdot x^{(d)}_{i_d} = \mathbf{B}_{i_1,\ldots,i_d}$ for all $i_1, \ldots, i_d \in \{1, \ldots, n\}$. In fact, a multidimensional version of the Sinkhorn–Knopp algorithm for scaling a matrix discussed previously can be used to obtain these d vectors.

1.5 Structure of the thesis

Chapter 2 examines the matching problem in bipartite graphs and consists of three parts. In the first part, we consider Karp–Sipser [72], a well-known matching heuristic. We describe an efficient implementation and investigate its worst-case complexity. We compare experimentally against its widely used simpler variant and show cases for which the full algorithm yields better performance. In the second part of the chapter, we examine two exact probabilistic matching algorithms for two special classes of bipartite graphs which are random 2-out graphs, and d-regular bipartite graphs. We generalize these two algorithms and turn them into practical heuristics for arbitrary bipartite graphs. The key element in these generalizations is matrix scaling, which we apply on the adjacency matrix of a given graph. Experimental results show that the heuristics are fast and obtain near optimal matchings. They are also more robust than the state of the art heuristics used in the cardinality matching algorithms, and are generally more useful as initialization routines. The third part of the chapter consists of a brief theoretical result about the derandomization of a known randomized heuristic for the bipartite matching problem [38]. We propose a way to replace the random decisions of the randomized algorithm with deterministic ones without reducing the approximation guarantee.

Chapter 3 examines the maximum matching problem in general undirected graphs. Similar to the previous chapter, we discuss efficient algorithms that can find large approximate matchings fast enough for the purpose of initialization. This chapter builds on the last two parts of Chapter 2 by considering algorithms based on matrix scaling. More specifically, the examined algorithms are based on the theory of 1-out bipartite graphs discussed by Dufossé et al. [38] and mentioned in Section 1.1. We extend the theory to hold for general undirected graphs. We show that our main heuristic has an approximation guarantee of around $0.866 - \log(n)/n$ for a graph with n vertices. We provide several experiments that verify the theoretical results.

Chapter 4 examines the problem of finding a maximum cardinality matching in a d-partite, d-uniform hypergraph. Because the problem is NP-Hard, we design several heuristics. Some of our heuristics are generalizations of known heuristics for the maximum cardinality matching problem in graphs. We also propose a novel heuristic based on tensor scaling to extend the matching via judicious hyperedge selections. This heuristic is inspired by the properties of matrix scaling for bipartite graphs, which we will see in Chapter 2. Experiments on random, synthetic and real-life hypergraphs show that this new heuristic has superior performance to the rest of the heuristics.

Chapter 5 investigates efficient randomized methods for approximating the number of perfect

matchings in bipartite graphs and general undirected graphs. The examined approach uses matrix scaling to sample a perfect matching from the graph and uses the probability of selection to return an approximation for the number of perfect matchings in the graph. This approach shares similarities with the scaling-based approach from Chapter 4 in the sense that scaling is applied at each step to model the current state of the graph as accurately as possible. The experimental analysis on random and real-life graphs shows improvements in the approximation over previous and similar methods from the literature.

Chapter 6 examines the Min-BvN-Decomp problem defined for doubly stochastic matrices in Section 1.3.2. We consider two known heuristics called $\textsc{Greedy}_{\text{BVN}}$ and Birkhoff for this problem. We show performance bounds for both of these heuristics. Our observations along with previous work [39] allow us to explain better why $\textsc{Greedy}_{\text{BVN}}$ has superior performance to Birkhoff. We then consider two modifications of $\textsc{Greedy}_{\text{BVN}}$ and demonstrate experimentally that these modifications can lead to a BvN decomposition with fewer matrices.

Chapter 7 concludes the thesis. We provide a summary with the most important points covered in each chapter and also discuss potential future work.

Chapter 2

Matchings in bipartite graphs

This chapter investigates the problem of matchings in bipartite graphs. We examine the problem from the perspective of matching heuristics. Our motivations were defined in Chapter 1 where we highlighted their importance in speeding up exact algorithms. We remind the overall approach in here. To find a matching of maximum cardinality efficiently for practical applications one can use a two-step process. In the first step, a *cheap* initialization heuristic or an approximation algorithm is used in order to quickly find a matching of large cardinality. This matching is then given as an input to the second step and improved via a relatively more expensive *exact* algorithm. It has been empirically shown that this strategy is much faster than running an exact algorithm from the beginning—see for example Langguth et al. [85].

Our work in this chapter can be separated into three parts. The first part examines a well-known matching heuristic due to Karp and Sipser [72]. The results of this part were published in the proceedings of the ALENEX 2020 conference [C2]. The heuristic applies whenever possible two deterministic rules. The standard implementation of the heuristic can have quadratic worst-case performance. Our theoretical contribution in this part is a subquadratic algorithm to apply the Karp–Sipser heuristic. In the experiments we show the potential gains of Karp–Sipser when tested against a simpler variant which applies only one of the two rules. In the second part we examine randomized matching heuristics which are based on matrix scaling. In this part our motivation is to produce robust and reliable heuristics that have near-optimal performance even in instances where other well-known heuristics might behave poorly. The results of this part were published in the proceedings of the ESA 2020 conference [C3]. Our motivation to use scaling follows from previous work [38] where scaling was applied to guarantee the existence of large matchings in random 1-out subgraphs. Initially, we consider two existing probabilistic algorithms for matching, which are optimal for two different classes of graphs. The first class is random 2-out graphs sampled from the complete bipartite graph. The second class is d-regular graphs where all vertices have degree equal to d. We adapt these two algorithms such that they base their random decisions on the values in the scaled adjacency matrix of the given graph. Experimentally, we demonstrate the efficiency and effectiveness of the two heuristics, while also demonstrating their usefulness as initialization methods in comparison with other heuristics. In the third part we discuss a deterministic approximation algorithm for matching. Here, we want to examine whether we can avoid augmenting path searches—usually employed by other deterministic algorithms—in order to surpass the 1/2 bound in approximation. This approximation algorithm is obtained through the derandomization of an earlier approximation algorithm [38] and again relies on matrix scaling.

The rest of the chapter is organized as follows. Section 2.1 provides a survey of known

heuristics and approximation algorithms for maximum cardinality matching. In Section 2.2 we investigate the Karp–Sipser heuristic. In more detail, Subsections 2.2.1– 2.2.4 are concerned with implementation issues, while Subsection 2.2.5 presents related some experiments. The second part of the chapter constitutes of Sections 2.3 and 2.4. In Subsections 2.3.1 and 2.3.2 we review the two existing randomized algorithms for the special classes of bipartite graphs and then adapt them for general bipartite graphs. In Subsection 2.3.3 we present the experimental results with them. The deterministic approximation from the third part is discussed in Section 2.4. The derandomization process is presented in Subsection 2.4.1, while some experiments with the method are given in Subsection 2.4.2. Section 2.5 summarizes the chapter and discusses some potential future work.

2.1 A survey on matching heuristics

The problem of finding a maximum cardinality matching can be solved in $\mathcal{O}(m\sqrt{n})$ and there exist efficient algorithms [62, 76, 97] for this purpose. In this section, we will present an overview of heuristics that can be used in the cheap initialization step.

Hopcroft and Karp's original algorithm [62] to find a maximum matching in a bipartite graph proceeds in phases. At each phase, it finds shortest augmenting paths, and augments the current matching along a maximal set of disjoint such paths. Each such phase runs in $\mathcal{O}(n+m)$ time. Stopping when the shortest augmenting paths are of length $2k+1$ at a phase no larger than k results in an $(1-1/(k+1))$-approximate matching and requires $\mathcal{O}(k(m+n))$ time in the worst case.

The Greedy [104] heuristic at each step chooses a random edge and matches the two end-points and discards both vertices and the edges incident on them. ModifiedGreedy [104] chooses a free vertex and then randomly matches it to one of its available neighbors. Another variant MinGreedy [104] (see also Magun [87] and Langguth et al. [85] for related algorithms) improves upon the ModifiedGreedy algorithm by selecting a random vertex of minimum degree. These Greedy-like algorithms obtain maximal matchings and are therefore 1/2 approximate. Slight improvements in the form of $1/2+\varepsilon$ were shown for these algorithms [5, 96], but there are theoretical bounds in the same vicinity [12].

The well-known Karp–Sipser heuristic [72] uses two deterministic rules to handle degree-1 and degree-2 vertices in an optimal way. If such vertices do not exist in the graph, a random decision is done akin to Greedy. While Karp–Sipser can be thought of as an improved version of Greedy-like algorithms, its approximation guarantee is similarly not greater than $1/2+\varepsilon$ [12]. Duff et al. [35] and Langguth et al. [76, 85] compared these algorithms for initialization in maximum cardinality matching algorithms and suggested using Karp–Sipser as initialization for general problems especially with the push-relabel based algorithms [76]. In addition, several theoretical papers [6, 23] analyzed the expected matching quality of the Karp–Sipser and derived variants for sparse random graphs.

In order to break the 1/2 barrier in approximation, one must consider a different form of randomization. The Ranking [73] algorithm achieves an approximation ratio of $1-1/e$, where e is the base of the natural logarithm. At the first step, it assigns a random priority to all columns. Then, it traverses over the rows and tries to match each row with its unmatched neighbor of largest priority.

The same approximation ratio is also achieved by a very simple parallel algorithm called OneSided [38]. The most involved step in the OneSided algorithm is the application of a

matrix scaling algorithm in the adjacency matrix $\mathbf{A}$ of the given bipartite graph. Once the doubly stochastic matrix $\mathbf{S} = \mathbf{RAC}$ has been obtained, each column selects randomly one of its neighboring rows and gets matched with it. More specifically, row r_i selects column c_j with probability s_{ij}. We will briefly discuss how to derandomize OneSided towards the end of this chapter. The same paper [38] proposed another randomized algorithm called TwoSided which extends OneSided and achieves a better theoretical guarantee. TwoSided allows both rows and columns to select randomly one of their neighbors, again basing the decisions on the values in the scaled matrix $\mathbf{S}$. The selections from both columns and rows give rise to a 1-out subgraph G_1 of the original host graph G. In the paper it was shown that G_1 has a maximum matching with expected cardinality at least 0.866 of the maximum in G. This bound is in fact tight for complete bipartite graphs. Finally, we note that additional information about these heuristics may be found in a recent survey [98].

2.2 An examination of the Karp–Sipser algorithm

As was discussed above, the Karp–Sipser heuristic (KS in short) empirically performs better than many, if not all heuristics, on average [35, 85]. In this section, we will focus on this heuristic and its implementation. The Karp–Sipser heuristic is based on performing reductions on a graph with no degree-0 vertices (they are discarded throughout); when a degree-1 or degree-2 vertex appears, Karp–Sipser reduces the problem to a smaller one via the following rules:

- **Rule-1**: At any time, if a degree-1 vertex u with neighbor v appears the edge $\{u, v\}$ is added to the matching and both vertices are removed from the graph. This decision is optimal in the sense that there exists at least one maximum cardinality matching in the current graph containing $\{u, v\}$.
- **Rule-2**: At any time, if there are no degree-1 vertices, and a degree-2 vertex u with neighbors v and w appears, u and its edges are 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 (excluding u). Karp and Sipser showed that a maximum cardinality matching for the reduced graph can be extended to obtain maximum cardinality matching for the original graph by matching u with either v or w depending on vw's match.
- When none of these rules can be applied, a random edge from the current graph is added to the matching. The two matched vertices as well as their adjacent edges are removed from the graph. This decision may not be optimal.

Both Rule-1 and Rule-2 have the property that they preserve $\mathcal{M}^*(G)$. We will use the notation $G^{(0)} = G$ to denote the initial graph, and $G^{(t)}$ to denote the graph after t random or rule-based decisions. Let $G^{(k)}$ be the first graph where neither Rule-1 nor Rule-2 is applicable. We call $G^{(k)}$ a *kernel* of G for the maximum cardinality matching problem, i.e., a reduced, smaller graph where one can obtain a maximum cardinality matching for G given a maximum cardinality matching for $G^{(k)}$ by following the reductions in reverse order. Thus, in addition to the already discussed initialization strategy, another use of the Karp–Sipser algorithm is to obtain $G^{(k)}$ initially, and then apply an exact algorithm on the smaller subgraph $G^{(k)}$, rather than G, which can significantly improve the performance. This technique is called *kernelization* [28] and is used in several problems [42, 84]. Note that obtaining $G^{(k)}$ can be computationally expensive.

While Rule-1 is simple to implement, Rule-2 is more complicated and requires more effort to be used in a matching heuristic. That is why in practice [85, 93], the Karp–Sipser heuristic

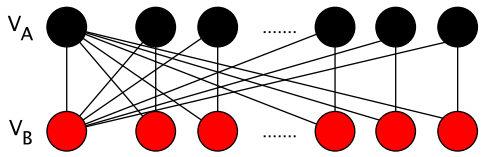


Figure 2.1 – A toy bipartite graph $G = (V_A \cup V_B, E)$ with vertex sets V_A and V_B. Except for the leftmost vertices from each vertex set, each vertex has two neighbors and will be removed by Rule-2 of Karp–Sipser. For every $G^{(t)}$, the merged vertices will have degrees $(n + 1 - t)$ and 2. Hence, the total time for the default Karp–Sipser implementation on this instance is $\Theta(n^2)$.

has usually been associated with its simpler variant KS_{R1}, which only applies Rule-1, and when necessary, random choices. Although it does not exploit the second rule, KS_{R1} has been shown to obtain large cardinality maximal matchings on real-life graphs, see e.g. [35, 76].

KS_{R1} can be implemented in $\mathcal{O}(n + m)$ time by keeping an up-to-date list of all degree-1 vertices. When the edge $\{u, v\}$ is added to the matching, if u is a degree-1 vertex, one needs to visit only v's neighbors to reduce their degrees. Otherwise, u's and v's neighbors must be visited and their degrees are updated. Hence, with only Rule-1, each adjacency list is accessed at most once and KS_{R1} runs in linear time. This essentially implies that the complexity of KS depends on the the cost of applying Rule-2.

For any graph $G^{(t)}$ that appears during the execution of the heuristic, let u be a degree-2 vertex with neighbors v and w. According to Rule-2, v and w must be merged to $vw \in G^{(t+1)}$. With a naive approach, this operation takes $\Theta(d_v + d_w)$ time. Since there can be at most n merge operations, the time complexity of this strategy is $\mathcal{O}(n^2)$. This bound is tight as one can create highly sparse graphs for which Karp–Sipser requires quadratic time as shown in Figure 2.1.

2.2.1 An expected $\mathcal{O}(m \log n)$-time algorithm

Let u be a degree-2 vertex with neighbors v and w. A Rule-2 application due to u operates in three steps:

1. Merging v's adjacency list with that of w,
2. Performing degree reductions for the vertices that are both v's and w's neighbors,
3. Updating the adjacency lists of the vertices which are neighbors of v and w.

Assume there exists an implementation of Karp–Sipser, hereafter referred to as KS_{min}, where it is possible to perform the 3-step merge operation in $\mathcal{O}(min(d_v, d_w))$ time. In the following theorem we show that the worst-case time complexity of KS_{min} is $\mathcal{O}(m \log n)$ which is better than $\mathcal{O}(n^2)$ for sparse graphs where $m \ll n^2$.

Theorem 2.1. KS_{min} *runs in* $\mathcal{O}(m \log n)$ *time.*

Proof. Rather than attempting to analyze KS_{min} directly, we will instead analyse a related hypothetical algorithm KS^*_{min} which operates on a multi-graph where multiple/parallel edges are allowed. In essence, KS^*_{min} never removes an edge from the graph and performs merges at any time step by simply concatenating the adjacency lists of the two vertices to be merged. KS^*_{min} will perform the same merges as KS_{min}. Note that in the multigraph of KS^*_{min} the corresponding vertices to be merged might not necessarily be neighbors of a degree-2 vertex (its

actual degree can be higher since no edges are deleted). Similar to KS_{min}, we assume that the cost of each merge in $\mathrm{KS}^*{}_{min}$ is equal to the size of the smaller adjacency list. Hence, for a single merge, the amortized cost per edge in the smaller list becomes $\mathcal{O}(1)$.

Let v and w be the vertices in multigraph $G^{(t)}$ and d_v, d_w with $d_v \leq d_w$ be the number of edges (parallel edges are allowed) incident on v and w. Let vw be the merged vertex in $G^{(t+1)}$. Note that d_{vw} is at most m, the total number of edges. We also have $2d_v \leq d_{vw}$, since we allow parallel edges. Therefore, a single edge can be in the smaller adjacency list in at most $\log m$ merge operations as otherwise d_{vw} would end up exceeding the number of edges m which is not possible as d_{vw} is bounded by m.

Having $\mathcal{O}(1)$ amortized complexity for each examined edge during a merge, the total time complexity of $\mathrm{KS}^*{}_{min}$ becomes $\mathcal{O}(m \log m)$. Since m is $\mathcal{O}(n^2)$, the complexity is $\mathcal{O}(m \log n)$.

To analyze the complexity of KS_{min}, we assume as we said the same merge operations between KS_{min} and $\mathrm{KS}^*{}_{min}$. Since the sizes of the merged adjacency lists in KS_{min} are always smaller than or equal to the corresponding lists in $\mathrm{KS}^*{}_{min}$, the time complexity of KS_{min} is also $\mathcal{O}(m \log n)$. □

It remains to discuss how to implement a merge operation on v and w in KS_{min}. Assume $d_v \leq d_w$. First, the larger list, i.e., w's list is kept intact and w becomes the merged vertex vw in the reduced graph. Then each neighbor x of v is processed one after another. To keep the graph simple and to correctly update the degree reductions for the common neighbors of v and w, x is first searched in the adjacency list of w. If x is already in w's list its degree is decremented and v is removed from x's adjacency list. Otherwise, x is inserted to w's list. Furthermore, v is also replaced by w in x's list.

To perform these operations in expected constant time per edge, we can use a hash table to store all the edges in the graph. This hash table is used to query the existence of x in w's list by looking whether $\{x, w\}$ exists or not in the hash table. With this data structure, we have $\mathcal{O}(m \log n)$ expected time complexity for KS_{min} and linear space. Instead of a hash table, one can use a data structure to store the edges with $\mathcal{O}(\log m)$ insertion, update and lookup time such as a binary search tree. Such a data structure yields an $\mathcal{O}(m \log^2 n)$ worst-case time complexity for KS_{min} in linear space.

2.2.2 An implementation with list caching

In order to merge the adjacency lists of two vertices v and w, one can use a dense, 0-1 array L of size n. This array represents v's adjacency list with $L[x] = 1$ iff $\{v, x\}$ exists in the current graph. Once such a list is created for one of the merging vertices, the edges of the other vertex can be looked up in L. Note that this array needs to be created every time there is a merge involving v and each time, one needs to re-iterate over v's adjacency list.

A natural optimization to this approach would be to allocate such arrays for some vertices and persistently keep them in memory, i.e., *cache* them. With caching, we do not have to re-iterate over these vertices' adjacency lists each time they participate in a merge. Indeed, if we cache two arrays for the two leftmost vertices in Figure 2.1, the complexity drops from $\Theta(n^2)$ to $\Theta(n)$. We refer to this variant as KS_{cache}.

There are many different strategies to decide on which vertices to keep in the cache. For example, one can cache the lists for the k most recently merged vertices, or k highest degree vertices. In Theorem 2.2, we show a negative result holding for any arbitrary caching strategy.

Theorem 2.2. KS_{cache} *using k arrays and applying Rule 1, Rule 2 and random decisions has an instance requiring $\Omega\left(n^2/k\right)$ time for all possible caching policies.*

Proof. Assume that we have enough memory to cache the adjacency lists of k vertices in dense form. Let G be the $n \times n$ bipartite graph shown in Figure 2.2 in which the vertices in V_A and V_B are shown in black and red, respectively. The graph contains $2k$ identical subgraphs, each having $\frac{\left(\frac{n}{2k}-1\right)}{2}$ two-by-two complete bipartite structures and an additional V_A (V_B) vertex that is connected to every other V_B (V_A) vertex in its subgraph. These extra vertices are labeled as a_i and b_i for the ith subgraph (in fact, they correspond to the left most vertices in Figure 2.1).

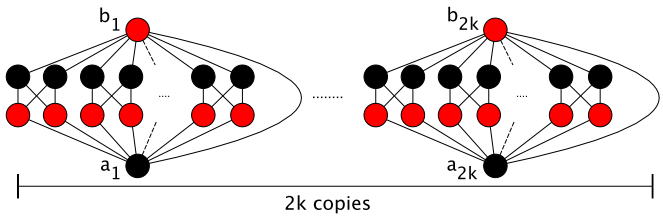


Figure 2.2 – A toy bipartite graph $G = (V_A \cup V_B, E)$ where $|V_A| = |V_B| = n$ and the vertices in V_A and V_B are colored with black and red, respectively.

As the figure shows, the minimum degree in the initial graph is three and a random decision is required. When a random decision hits into a two-by-two part in the ith subgraph, the corresponding matching and the removal of the matched vertices yield two possible merges involving either a_i or b_i. These potential merges eliminate each other, i.e., only one of them is possible. Furthermore, after the merge operation, the minimum degree in the graph will go back to three. Hence, assuming all the random edges are from the two-by-two bipartite subgraphs, each random choice is followed by a merge which is then followed by another random decision. Since the decisions are random, we cannot build a caching strategy based on them.

Hence, regardless of caching policy, it is always possible that the neighbors of neither a_i nor b_i are in the cache until at least k components are reduced completely. Doing so for only one component without considering the amount of work for the others takes

$$\sum_{j=0}^{\left(\frac{n}{2k}-1\right)/2} \left(\frac{n}{2k} - 2j\right) = \Theta\left(\frac{n^2}{k^2}\right)$$

time since the degrees of a_i (as well as b_i) vertices are reduced by two after each step. Since at least k components must be consumed, the complexity until this point is $\Omega\left(n^2/k\right)$ which implies a lower bound on the execution time of KS_{cache} for the instance in Figure 2.2 assuming that the random choices always hit two-by-two subgraphs. □

2.2.3 An alternating component approach

Here, we consider and extend an idea that was first introduced by Langguth et al. [85]. There, the authors used the term *alternating component* to refer to a connected subgraph consisting of a set of *boundary vertices* connected by paths. The boundary vertices are vertices of arbitrary degree greater than one. The paths are required to contain an even number of edges, have a boundary vertex as beginning and endpoint, and all other vertices in each path are required to

be of degree 2. An example can be seen in Figure 2.3. In addition, we only consider components whose paths are maximal, i.e., they cannot be extended without violating the above definition.

If an alternating component contains a cycle, then there is no need to explicitly merge vertices in the component. Since every edge has at least one incident vertex of degree 2, at least half of the vertices in such a cycle have degree 2. Thus, for any edge of the cycle, there is a maximum cardinality matching containing that edge. Thus, we can pick an arbitrary edge and match it. The remaining component will be matched via the application of Rule-1.

If a component is acyclic, it can be immediately merged into a single vertex via Rule-2. To see this, consider two vertices that are both connected to a degree-2 vertex in a simple alternating component. Thus, each application of Rule-2 will reduce the length of a path between boundary vertices in an alternating component by two, and because the length is even, they will be merged when it reaches zero. The same applies to any such path in an alternating component. The advantage of doing so is that we do not have to perform merges immediately. Instead, we maintain components whose merges can then be performed together more efficiently.

Langguth et al. [85] applied only the first operation. Acyclic components were maintained, but not explicitly merged. This resulted in a useful heuristic, but it is not equivalent to a full implementation of Rule-2. Here, we show how to use alternating components to implement the Karp–Sipser algorithm in its entirety. We will call this version as KS_{comp}.

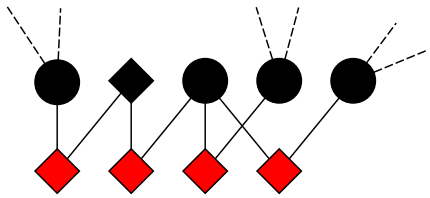


Figure 2.3 – An example of a component: diamond shaped vertices correspond to the degree-2 vertices and cycle shaped vertices correspond to boundary vertices. The dashed lines correspond to additional edges, unrelated to the displayed component.

Similar to the basic implementation, the algorithm keeps stacks of degree-1 and degree-2 vertices. If a degree-1 vertex exists at any time in the remaining graph, we add its edge to the matching. For a degree-2 vertex u, we create $P(u)$, a maximal even-length path containing u and only degree-2 vertices except for the first and last vertex in $P(u)$. Let l, r be these boundary vertices in $P(u)$. We use a union-find data structure to keep track of membership in components. At the start of the algorithm, every vertex is its own component. Now, if l and r already belong to the same component C, then adding the additional path must close a cycle in C. In that case we arbitrarily add one of the edges of u to the matching and the component is then cleared out by Rule-1 reductions. If no cycle exists, then we simply update the labels of l, r to signal that their components have been joined together. No merge is performed at this point.

When there are no degree-2 vertices left, we merge the remaining acyclic components one by one. We refer to this as a *component shrink*. Let C be component with boundary vertices $b_1, \ldots, b_k$. To perform the shrink we require $\mathcal{O}(\sum_{b_i \in C} d_{b_i})$ time since we only need to access the adjacency list of each b_i once. If during these merges, a new degree-2 vertex becomes available (this can happen for example if b_i and b_j are both adjacent to some vertex w with $d(w) = 3$ before the shrink) the merging stops and the degree-2 operation is applied, which can

unify two components. If all components have been shrunk, and no degree-1 or degree-2 vertices exists, KS_{comp} selects a random edge to be matched as in the standard case.

Using alternating components can prove useful in the sense that they provide a structured way to perform reductions such that they avoid re-reading adjacency lists of vertices (which was also the motivation of KS_{cache}) . For example, it takes only linear time for the worst case instance of Figure 2.1. However, it can degrade to the worst case performance of the basic implementation if there are no nontrivial alternating components in the graph.

Theorem 2.3. *The* KS_{comp} *variant requires* $\mathcal{O}(n^2)$ *in the worst case.*

Proof. The proof is similar to that of Theorem 2.2 hence we skip the details. In short, if the algorithm is given an input instance as in Figure 2.2, there will be no non-trivial components. Assuming all random choices hit into the two-by-two complete bipartite subgraphs, there will always be a random choice following a single merge. Hence, on this instance, the component approach behaves just like the basic implementation. □

We note that the approach of using alternating components bears some similarity with the approach used in [7], which performs all currently existing Rule-2 operations in linear time. However, the authors consider Rule-2 exclusively and their approach cannot handle Rule-1 or random decisions. They propose a complicated static linear-time algorithm which operates on the DFS-tree of a given graph and avoids to do merges explicitly by doing implied reductions. Let us refer to a *phase*, as a series of consecutive Rule-1 or random operations done by KS. Phases are thus separated from each other by one or more Rule-2 applications. Their $\mathcal{O}(n+m)$ time algorithm would need to be re-run between the different phases and thus can potentially yield a complexity of $\mathcal{O}(n^2)$.

2.2.4 Fast recovery of the matching

As described above, a merge operation creates a new vertex that may be involved in upcoming merges. Hence, the matched vertices returned as the final output by the heuristic do not necessarily appear in $G^{(0)}$. The standard way to recover the actual matching is by keeping a stack S containing all the information related to each merge [87]. After the heuristic has finished, the stored merges are popped one after another from S in order to be create a matching for $G^{(0)}$. Assume that during the heuristic a degree-2 vertex u appears with neighbors v and w; hence, a new vertex vw is created. While recovering the actual matching, if vw is unmatched we can match either v or w with u and continue. Otherwise, we need to check whether vw is matched with one of v's or w's neighbors stored in S. Using similar reasoning as in Theorem 2.1, this approach takes $\mathcal{O}(m \log n)$ time and space in the worst case, assuming the smaller neighborhood (among v's and w's) is stored in the stack.

Here, we propose an algorithm with $\mathcal{O}(n)$ time complexity. To recover the matching efficiently, we use a graph $F \subseteq G$. Initially, F has no edges but contains all vertices of G. During the heuristic, assume a merge is being performed due to a vertex having two edges e and e'. Let $\{u, v\}$ and $\{u', w\}$ be the original endpoints of e and e' in G, respectively. We add the edges $e_1 = \{u, v\}$ and $e_2 = \{u', w\}$ to F and set $twin(e_1) = e_2$ and $twin(e_2) = e_1$. The association of e_1 and e_2 as twins exploits the implicit Rule-2 property that either e_1 or e_2 can be in a maximum cardinality matching, but not both.

Proposition 2.1. *F is a forest.*

Proof. Assume that F contains a cycle. Let $C \subseteq F \subseteq G$ be a simple cycle with length ℓ. Let C be first initiated with a merge in $G^{(t)}$, and let u be the degree-2 vertex for that merge with neighbors v and w. Both $\{u, v\}$ and its twin $\{u, w\}$ must be in C, since u will not exist in $G^{(t+1)}$. Let C' be the reduced cycle obtained by removing u from C and merging v and w. The above steps can be repeated by using the edge that initiates C'. At each step, the cycle will be further reduced and after $l - 1$ steps, only two vertices and a single edge will remain. As parallel edges are not allowed in the graph $G^{(\cdot)}$, such a merge cannot exist to complete the cycle. □

Let M' be the set of matched edges found by the heuristic and let M be the the matching constructed with the original versions of M''s edges from $G^{(0)}$. Let $e = \{u, w\}$ be an edge in M. Then for all edges $e' \neq e$ of u and $e'' \neq e$ of w in F, we add $e^* = twin(e')$ and $e^{**} = twin(e'')$ to M. We then remove u and w from F (i.e., $F = F \setminus \{u, w\}$) and consider the next matched edge in M. If no more nodes in the current tree are matched, we select a vertex of degree-1 in F, add its unique edge to M and repeat.

Lemma 2.1. *The algorithm described above to recover the matching is correct.*

Proof. Let e and e' be two twin edges which have not been included in the matching M. If one of the endpoints of e is matched in M, then it is no longer possible to include e in M. By Rule-2, one of the twins must necessarily participate in a maximum cardinality matching. Hence, we can extend M by using e's twin edge. This means that as long as there exist matched vertices in F, the algorithm correctly extends the matching.

If there are no matched vertices in F, for any twins e and e', we know that either can be in a maximum cardinality matching. Hence, we can arbitrarily add either to M. For simplicity, we chose the edge of a degree-1 node, which always exists since F is always a forest. □

2.2.5 Experiments

We implemented the algorithms discussed in the last section. For convenience, we use the shortened form KS to refer to Karp–Sipser throughout the experiments. Except where noted, the KS algorithm variant being used is the default one.

When KS is used as a heuristic, we first generate a random permutation of the edges with uniform probability, which we store in a list. Then, in the event of a random decision, the first available edge (that is without any matched endpoints) is returned from this list.

All the codes are written in `C++` and compiled with `gcc 8.3.0` with `-O3` optimization flag. They are tested on two different machines. The first machine had 2 x AMD EPYC 7551 CPUs and 256 GB RAM (Arch 1). The other had 4 x Intel Xeon E7-8890 CPUs and 1.5 TB RAM (Arch 2).

We used a vector-based implementation for both KS and KS_{R1}, because we wanted to measure exactly the additional costs incurred due to the addition of Rule-2. When comparing between KS and KS_{R1} we thus do not include costs such as initialization or randomization which are identical for both.

2.2.5.1 On real-life graphs

Our real-life dataset consisted of 93 bipartite graphs with $n = |V_R| = |V_C|$ and $10^6 \leq n \leq 10^7$ taken from the SuiteSparse Matrix Collection [31]. The number of edges in these graphs varied from $3 \cdot 10^6$ to $3.1 \cdot 10^8$. For each of these matrices, we performed a total of sixteen runs. More specifically, we randomly permuted each matrix four times, and then measured the timings for

both KS_{R1} and the default KS algorithm four times in the permuted matrix. Permuting the matrix between runs can potentially affect the run-time for both KS heuristics as well as the exact algorithm used afterwards. To analyze the impact of adding Rule-2 for data reduction, two exact matching algorithms push-relabel (PR) [55] and Pothen-Fan+ (PF+) [35, 97] are used. In previous studies [76], these two algorithms are shown to be the best performing algorithms from a set of alternatives, including the asymptotically fastest one [62].

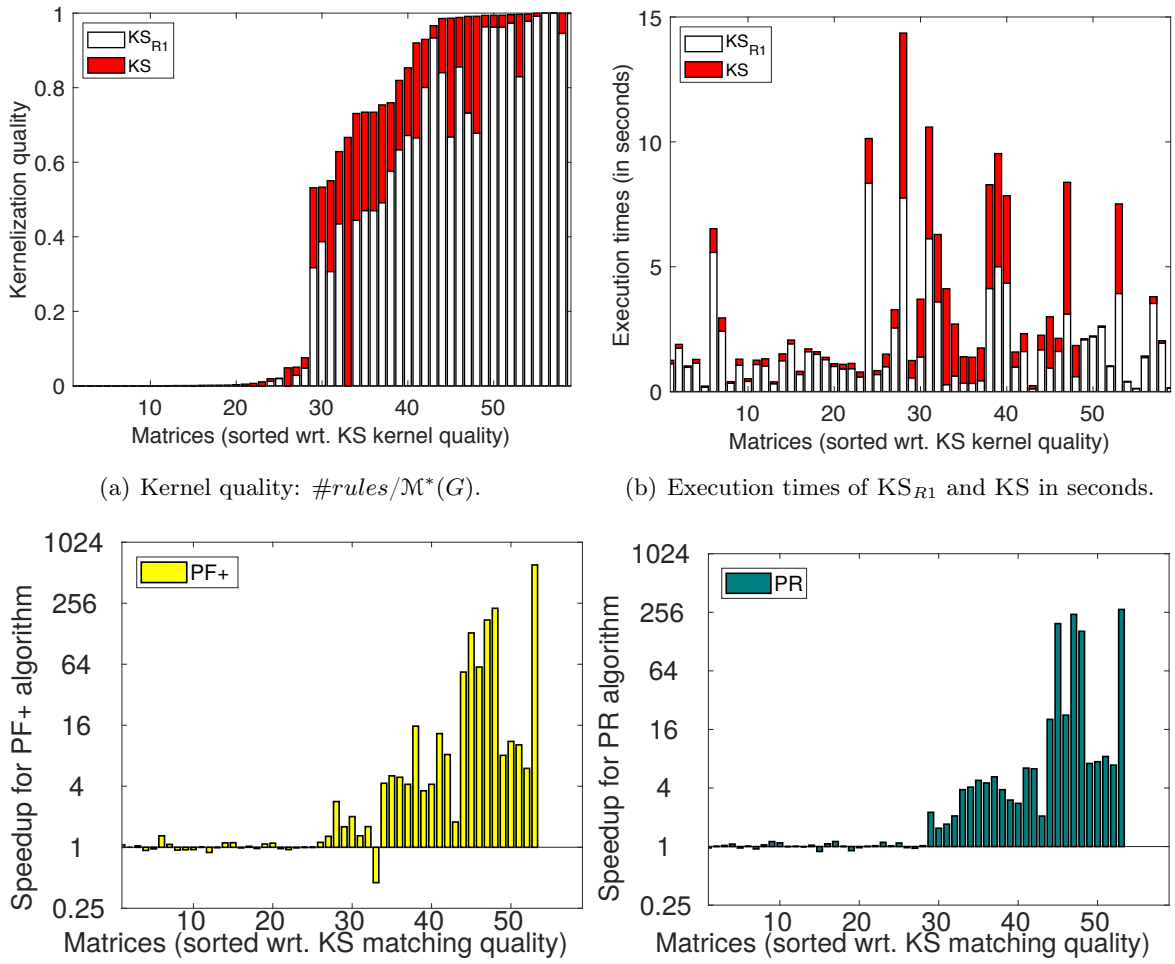
Using KS_{R1} for kernelization. We first consider the use of KS as a kernelization tool and run experiments on Arch 1: we first run KS and KS_{R1}, with no random decisions, to find their kernels $G^{(k)}$ and $G^{(k')}$ where $k \geq k'$. After that, an exact algorithm, PR or PF+, is executed to find a maximum cardinality matching in the kernel. As KS requires a minimum degree at most 2 in order to return a non-trivial kernel, for this experiment, we only use the graphs having min degree one or two. There are 59 such graphs. The results are summarized in Figure 2.4.

To measure the impact of the second rule, we measured the kernel quality for each heuristic. The quality is computed as the number of times a rule is applied during KS and KS_{R1}. normalized with respect to size of the maximum cardinality matching. Figure 2.4(a) shows that in terms of quality, for around 30 matrices, the second rule has an impact and for one matrix, it increased the quality from almost 0 to 0.65. Hence, KS obtains a kernel of smaller size compared to KS_{R1}. Furthermore, as Figure 2.4(b) shows, KS is slightly slower compared to KS_{R1} differing in most graphs by less than a second.

To evaluate the impact of additional size reduction obtained via Rule 2 on the exact matching process, we measured the execution times of PR and PF+ given the kernels $G^{(k)}$ and $G^{(k')}$ obtained by KS and KS_{R1}. Figures 2.4(c) and 2.4(d) shows the speedup obtained, i.e., the execution time of PF+ or PR on $G^{(k')}$ divided by the execution time of the same algorithm on $G^{(k)}$. In these two figures, if the speedup is higher than one, then it is in favor of KS. The first observation is that the speedups are mostly in the positive side and a good kernel can significantly improve the execution time of the matching phase. The second observation is that the impact of kernelization heavily depends on the algorithm used since PR and PF+ can behave in a different way. For instance, in one matrix, although $G^{(k)}$ is smaller than $G^{(k')}$, PF+ runs slower on $G^{(k)}$. Yet, even with such fluctuations, for both algorithms, smaller kernels usually yield better performance.

Using KS as a heuristic. To check if KS is also useful as an initial, cheap matching heuristic, we let KS and KS_{R1} run in their entirety and input the returned matchings to the exact algorithms. The experiments are performed on Arch 2. Figure 2.5 summarizes the results of these experiments. Both KS_{R1} and KS obtain excellent results with matching quality almost always larger than 0.97 of $\mathcal{M}^*(G)$. KS is able to match 1% more vertices than KS_{R1}, which corresponds to more than ten thousand vertices due to the size of our instances.

As in the previous experiments, the exact algorithms tend to be faster when initialized with KS. However, since the impact, i.e., the difference in the quality, is higher for kernelization compared to matching, the speedups are not as large as the previous set of experiments. If we look at the total run time to find a maximum cardinality matching, we see that on average using KS rather than KS_{R1} can yield a 20% slow-down using PF+ and 40% with PF+. This is understandable since KS itself does significantly more work than its simpler variant. On the other hand there exist instances where using KS rather than KS_{R1} can be crucial. For example, consider the graph `com-LiveJournal` for which PF+, initialized with KS_{R1} matching, results in a run time of about 1060 seconds on average, whereas PF+ requires 65 seconds on average, if



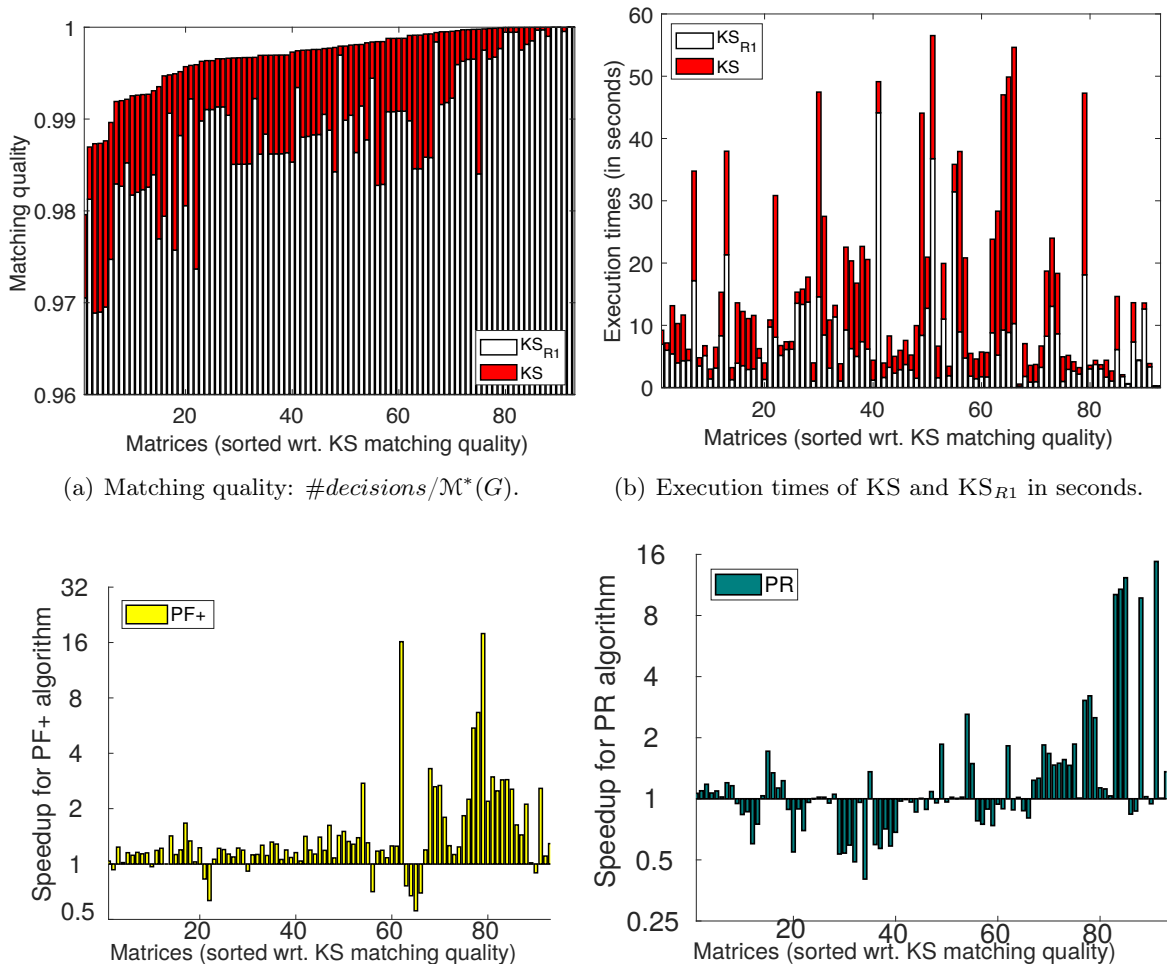
(a) Kernel quality: $\#rules/\mathcal{M}^*(G)$.

(b) Execution times of KS_{R1} and KS in seconds.

(c) Speedups for PF+ when the KS kernel is used instead of KS_{R1} kernel.

(d) Speedups for PR when the KS kernel is used instead of KS_{R1} kernel.

Figure 2.4 – (a) The kernel quality of KS and KS_{R1} for the 59/93 graphs having at least a single vertex with degree strictly less than three. The quality is measured as the number of rules applied during a heuristic normalized with respect to size of the maximum cardinality matching. (b) The execution times (in sec.) of KS_{R1} and KS. The speedups for the exact matching algorithms PF+ in (c) and PR in (d) when the KS kernel is used instead of KS_{R1}. In all figures, the matrices in the x-axis are listed with respect to non-decreasing kernel quality for KS.



(a) Matching quality: $\#decisions/\mathcal{M}^*(G)$.

(b) Execution times of KS and KS$_{R1}$ in seconds.

(c) Speedups for PF+ when KS is used as the initial heuristic instead of KS$_{R1}$.

(d) Speedups for PR when KS is used as the initial heuristic instead of KS$_{R1}$.

Figure 2.5 – (a) The matching quality of KS and KS$_{R1}$ for all 93 graphs. The quality is measured as the number of decisions, including random ones, applied during a heuristic normalized with respect to size of the maximum cardinality matching. (b) The execution times (in sec.) of these two heuristics are also shown. The speedups for the exact matching algorithms PF+ in (c) and PR in (d) when KS is used instead of KS$_{R1}$. In all figures, the matrices in the x-axis are listed with respect to non-decreasing matching quality for KS.

	KS_{R1}				KS	
n	quality	time	PF+ (in seconds)	PR (in seconds)	quality	time (in seconds)
7500	0.75	1.01	1.29	1.30	1	0.73
10000	0.75	1.95	2.86	2.57	1	1.46
15000	0.74	5.61	7.48	6.37	1	4.48
20000	0.74	10.34	17.08	12.37	1	8.82

Table 2.1 – The average quality and timings of the heuristics on the family of graphs $\mathcal{J}$ referenced in Section 2.2.5 for $n \in \{7500, 10000, 15000, 20000\}$ as well as the timings of PF+ and PR when initialized with KS_{R1}'s input. For these graphs, KS finds a perfect matching. The PF+ and PR timings do not include the time needed for KS_{R1}.

initialized with KS. That is why we are also interested in parallel implementations of Rule-1 and Rule-2 which will potentially bring the performance of KS closer to that of KS_{R1} and hence reap more benefits from the speedups seen in Figures 2.5(c) and 2.5(d). In both figures a value higher than one is in favor of KS.

2.2.5.2 On synthetic datasets

Here we focus on special, synthetic instances, for which Rule-2 increases the probability of finding a maximum cardinality matching. Anastos and Frieze [4] have described additional classes of graphs where Rule-2 also helps to find a perfect matching.

Optimal instances for KS_{R1}. The first dataset is a family of synthetic graphs where Rule-2 is sufficient to find a maximum cardinality matching. We will use $\mathcal{J}$ to refer to graphs of this family. Let $G = (V_R \cup V_C, E)$ be a bipartite graph with $|V_R| = |V_C| = n$. For any $1 \leq i \leq j \leq n$, there is the edge $\{r_i, c_j\}$ in G where $r_i \in V_R$ and $c_j \in V_C$. The edges $\{r_2, c_1\}$ and $\{r_n, c_{n-1}\}$ also exist.

In the graph, the vertices $r_{n-1}, r_n \in V_R$ and $c_1, c_2 \in V_C$ have degree equal to two. Assume without loss of generality that we apply Rule-2 on c_1, to merge r_1 and r_2. Then in the reduced graph, $r_2 r_1$ will be a degree-1 vertex hence Rule-1 can now be applied. We continue with Rule-1 until four vertices remain; applying Rule-2 followed by Rule-1 yields a perfect matching. Hence KS with Rule-2 always find a perfect matching for these synthetic graphs. Furthermore, since Rule-2 is applied only twice, KS requires linear time. If however, only Rule-1 reductions are allowed, no reduction is possible at first and KS_{R1} will immediately resort to random decisions, which understandably affect negatively its performance. In Table 2.1, we provide the average quality returned by KS_{R1} for different values of n as well as run times on the synthetic graphs of family $\mathcal{J}$. As can be seen, there is 25% quality difference in favor of KS. Interestingly, KS_{R1} requires slightly more time than KS. This can be explained by the fact that due to the random decisions, during most steps KS_{R1} needs to iterate over two adjacency lists, while KS needs to iterate over only one, due to it applying Rule-1 almost always. Furthermore, the cost of both exact algorithms increases with n.

Results with random 2-out bipartite graphs. We examine the behavior of KS on random 2-out graphs sampled from the complete bipartite graph. Note that in the following section we will discuss an exact algorithm for such graphs.

By construction, each vertex in a 2-out graph has degree at least two and therefore KS_{R1} will resort to random decisions immediately. This is not the case however for KS. In expectation,

n	KS_{R1}	KS
10000	0	0.68
25000	0	0.64
50000	0	0.56

Table 2.2 – The ratio of tests for which KS_{R1} and KS find a perfect matching in random 2-out graphs.

n	KS_{R1}	KS	KS_{comp}
40000	0.00	3.44	0.04
80000	0.01	15.32	0.11
160000	0.03	64.53	0.22
320000	0.10	265.89	0.48

Table 2.3 – The run time in seconds for KS_{R1}, KS and KS_{comp} in the worst-case instance shown in Figure 2.1 for $n \in \{40000, 80000, 160000, 320000\}$.

a random 2-out graph will initially have about $\frac{2n}{e^2}$ degree-2 vertices. Hence Rule-2 can potentially be applied a significant number of times before the graph runs out of degree-2 vertices. Consequently, because it has a smaller kernel and can apply its rules easier, KS will have to make random decisions less frequently than KS_{R1} and therefore has less of a chance to make an erroneous decision.

In Table 2.2, we compare KS_{R1} and KS on 2-out graphs of varying size n and show the percentage of tests where the two algorithms found a perfect matching. For each $n \in \{10000, 25000, 50000\}$, we generate 20 different random 2-out graphs. Each distinct random 2-out graph is also used four times to allow alternative random decisions on different runs. As the table shows, KS_{R1} is never able to find a perfect matching on such graphs. On the other hand, KS is able to find one in a significant percentage of the trials. Nonetheless, both algorithms output matchings of high quality i.e., greater than 0.99. In addition, both KS and KS_{R1} require less than a second in all of the instances.

Experiments with worst-case run time inputs. In the results with the real graphs for Section 2.2.5.1, we observed that KS behaves in practice very similar to a linear time algorithm. No instance led to quadratic behavior. For this reason, the various modifications discussed in Section 2.2 in most cases did not have a noticeable positive impact on the performance.

However, we do note that our modifications can indeed be useful and we demonstrate in Table 2.3 the performance for various values of n for the instance in Figure 2.1. For simplicity, we provide only results with KS_{cache} but an implementation of KS_{min} using a hash table, and KS_{comp} both had equivalent performance. As can be seen, while KS_{R1} and KS_{comp} require in all instances less than a second, KS struggles as n increases. For example, it requires over four minutes in the last case. This shows that in certain situations the sub-quadratic algorithms of Subsection 2.2.1 might be crucial due to their reduced worst-case behavior. Similarly the other algorithms in Section 2.2 might prove useful as they can avoid some pitfalls that harm the performance of KS.

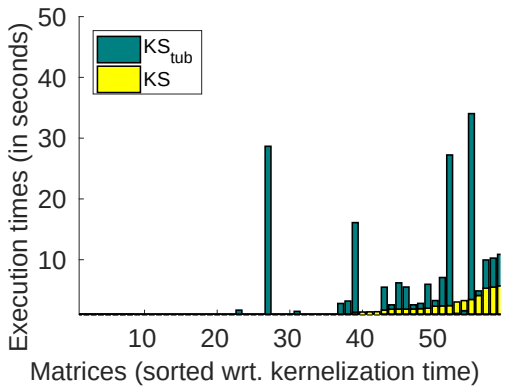
2.2.5.3 Comparison with another implementation

Korenwein et al. [82] provide an implementation in C++ which finds the kernel $G^{(k)}$ of a given undirected graph G. We use KS_{TUB} to denote this code. The code KS_{TUB} works along the same lines as our kernelization code. Initially, Rule-1 or Rule-2 are applied for as long as it is possible, and the kernel $G^{(k)}$ is created. Then, the vertices in $G^{(k)}$ are renumbered to be from 1 until $|V_{G^{(k)}}|$. This smaller representation of the kernel is then given as input for the exact matching algorithm. The code KS_{TUB} was compiled according to the instructions.

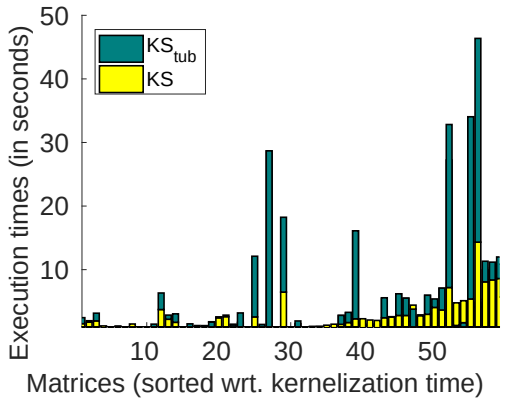
We now list some of the differences between our implementation and KS_{TUB} before giving experimental results. One important difference between our implementation and KS_{TUB} is that we prioritize Rule-1 over Rule-2. In KS_{TUB}, vertices of degree one and two are stored together in a container, and the appropriate reduction rule is executed when a vertex is accessed from this container, depending on its degree. As a consequence, KS_{TUB} might apply Rule-2 more frequently, which can harm the performance. Indeed, consider an extension of the graph shown in Figure 2.1, in which the two leftmost vertices are connected with an additional degree-1 vertex each. Then, by applying Rule-1 on one of the new vertices, it becomes possible to find the kernel in linear time. If, however, the application of Rule-1 is postponed in favor of Rule-2, then the worst case run time becomes quadratic. The code KS_{TUB} also uses linked lists which are not as cache-friendly as our vector-based implementation. Furthermore, our implementation also returns a matching for G, whereas KS_{TUB} returns only the kernel and does not return any information about which edges or vertices of G have been involved in the Rule-2 reductions. Hence, it is not possible to extend the matching found in the kernel $G^{(k)}$ to a maximum cardinality matching for G using KS_{TUB} as is.

We now give the run time comparison of the two codes. The experiments were conducted on Arch 1. For the sake of controlled experimentation, we made the two codes have the same permutation of the input graphs. KS_{TUB} normally prints output; we removed this part for fairness. In Figure 2.6, we compare the two codes on the real-life bipartite graphs used in the kernelization experiments from Section 2.2.5. For each implementation, we report the minimum time observed over four runs. In both implementations, we measured the time to apply Rule-1 and Rule-2 and the time needed to renumber the vertices in $G^{(k)}$. In addition, for our implementation, we included additional costs such as the initial cost required to create the adjacency list representation of G and as well as the time needed by the algorithm of Section 2.2.4 to find a maximum cardinality matching of the original graph (by using and expanding the one on the kernel to comply with the Rule-2 applications). As can be seen, our implementation is almost in all cases faster than KS_{TUB} and in some situations significantly so. In regards to the time required solely for the purpose of kernelization, shown in Figure 2.6(a), we see that our implementation is about 3.19 times faster on average than KS_{TUB}. The maximum time our code requires is 5.8 seconds, whereas KS_{TUB} exceeds 10 seconds on six instances. The difference in speed becomes more noticeable in instances such as `kron_g500-logn21`, where our implementation takes 3 seconds, while KS_{TUB} takes 34 seconds.

The difference needed to renumber the vertices in the kernel can similarly be as striking as can be observed in Figure 2.6(b). For example, on the bipartite graph `Com-Orkut`, our code to renumber vertices takes 5.6 seconds, whereas KS_{TUB} requires 41 seconds. We estimate that this significant difference derives from the fact that KS_{TUB} creates an entirely new class object, for which it then has to initialize a number of linked lists (one per vertex). In contrast, our code needs to initialize only four arrays as $G^{(k)}$ is recreated in the standard CSR/CSC formats of sparse matrices to be given as input to the exact algorithms.



(a) Comparison only for kernelization between our implementation of KS and KS_{TUB}.



(b) Overall time comparison between our implementation of KS and KS_{TUB}.

Figure 2.6 – Comparison of our implementation with KS_{TUB} [82] on the graphs of Section 2.2.5.1. Figure 2.6(a) gives only the time needed to find the kernel by applying Rule-1 and Rule-2. Figure 2.6(b) gives the overall time by adding the time needed to renumber the vertices of the kernel $G^{(k)}$. In addition, for our implementation Figure 2.6(b) also includes the time required to recover the maximum matching using the algorithm of Section 2.2.4, and the time to create the adjacency list representation for G.

Finally, looking on the overall time, again in Figure 2.6(b), our implementation is about 2.17 times faster than KS_{TUB} even with recovery of a maximum cardinality matching for the initial graph. Our implementation never exceeds 15 seconds in overall time, whereas there exist six graphs where KS_{TUB} exceeds this time bound.

2.2.6 Related work

To the best of our knowledge, there are only a few studies focusing on implementing Karp–Sipser as in its original, proposed form. Magun [87] claims an implementation with linear run time for Karp–Sipser but from the pseudocode and description in the paper, it is not clear whether the reductions are implemented in a way that is equivalent to ours. Analysis of the source code however does suggest that the implementation is equivalent and hence can similarly require quadratic time. The paper also does not study practical running time. Mertzios et al. [90] apply the first reduction rule and a restricted version of the second rule described before in linear time to obtain a linear-size kernel with respect to a graph parameter (known as feedback edge number) for computing maximum cardinality matchings in undirected graphs. A cubic kernel on bipartite graphs with the same reduction rules is also shown. Korenwein et al. [82] discuss extensions of two rules due to Karp and Sipser for the maximum weighted matching problem. Their results indicate that kernelization is less successful in the weighted case in comparison to the unweighted case. Bartha and Krezs [7] discuss a linear time algorithm where only Rule-2 is considered.

2.3 Scaling based near-optimal randomized algorithms

On the second part of the chapter, we will examine two randomized matching heuristics which make use of matrix scaling. These heuristics generalize two existing randomized algorithms [54, 71], based on two classes of randomized algorithms called *Las Vegas* and *Monte Carlo* algorithms [92, p. 70]. For convenience, we remind here the definition of these two classes. Las Vegas algorithms always return a correct answer, but their run time can depend on random choices, whereas Monte Carlo algorithms can fail with small probability, but their complexity is independent of the random choices made.

The Monte Carlo algorithm due to Karp et al. [71] finds almost surely, maximum cardinality matchings on random graphs formed by allowing each vertex to select any two vertices from the other side uniformly at random. The Las Vegas algorithm due to Goel et al. [54] finds maximum cardinality matchings in regular bipartite graphs, where all vertices have equal degree. In both of these classes of graphs, the bipartite graphs have equal number of vertices in each part, and the maximum cardinality matchings cover all vertices (i.e., they are perfect). Both algorithms work in $\mathcal{O}(m + n \log n)$ time.

We generalize these two exact algorithms taking inspiration from the ideas explored by the TwoSided and OneSided algorithms discussed in Section 2.1. We scale the adjacency matrix of the input bipartite graph and use the nonzero values of the scaled matrix for sampling, i.e., for choosing randomly a neighbor for a given row or column. Both produced heuristics run in near linear time and obtain matchings whose cardinality is more than 0.99 of the maximum, even in cases where the current state of the art approaches have difficulties. We also identify and fix an oversight in the Monte Carlo algorithm.

2.3.1 2OUTMC: Monte Carlo on 2-out graphs

2.3.1.1 Description of the algorithm

The Monte Carlo algorithm by Karp et al. [71] finds a perfect matching, with high probability, in a random 2-out bipartite graph, sampled from the complete bipartite graph. A random 2-out bipartite graph B_{2o} is constructed by selecting uniformly at random two row vertices for each column, and two column vertices for each row. These selections form the edges of B_{2o}. Given the edges of B_{2o}, Karp et al. define two multigraphs. The multigraph *Column-Graph* (CG) is the multigraph whose vertices are the rows, and whose edges are the choices of the columns. That is, there is an edge in CG for a column vertex c_u in B_{2o} and this edge will connect the two row vertices r_1, r_2 that c_u chose during the 2-out selection process. Parallel edges are allowed and can occur if for example two columns select the same two rows. The *Row-Graph* (RG) is defined similarly with rows as vertices and columns as edges.

The main idea to show that B_{2o} has a perfect matching is the following. In a component of CG that contains a cycle, it is possible to match all rows (vertices in CG) with one of the columns that have selected them (edges in CG). On the other hand in a tree component of CG, in any matching (pairing of edges with vertices) there will always be a free row vertex. As a consequence, when one or more trees appear in CG, the choices of the columns alone do not suffice to find a perfect matching, and those of the rows must be used. The algorithm thus keeps track of the tree components of CG and tries to identify one row vertex per tree component whose selections should be taken into account. The columns selected by such a row could be used for a set of rows belonging in tree components. Thus one should go back and forth identifying trees in CG and analyzing components in RG. Karp et al.'s algorithm, which is described in Algorithm 2.1, formalizes this approach.

The algorithm operates on H_1, a copy of CG, and H_2, a copy of RG initially devoid of edges. In each step, it picks a tree from H_1 and marks one of its vertices x. This signifies that x can only be matched with one of its choices. Then, the edge of x is inserted in H_2. The algorithm then finds the component Q_x in H_2 containing the edge x, and selects an unchecked column y from Q_x. Column y is checked, which means that it can only be matched with a marked vertex. As y's choices are rendered useless now, the corresponding edge is removed from H_1 upon which new trees can arise. For each tree vertex x identified in H_1, one should be able to find a vertex in the associated component Q_x, so that x can be matched in that component. Otherwise, Q_x has more edges than vertices, and any matching will leave some edges unpaired. The algorithm returns failure upon detecting this case (Line 10). The algorithm terminates successfully if all trees have a marked vertex. If this happens, each component in H_1 will have as many edges as unmarked vertices. Likewise, each component in H_2 will have as many edges as checked vertices. It is therefore possible to orient the edges in either H_1 or H_2 such that each vertex (excluding marked rows or unchecked columns) is matched with a unique adjacent edge. This gives a perfect matching in B_{2o}, which can be found by the Karp–Sipser heuristic in linear time. Algorithm 2.1 finds a perfect matching with probability $1 - \mathcal{O}(n^{-\alpha})$, where α is a positive constant. The authors then describe how to efficiently implement the algorithm such that it runs in $\mathcal{O}(n \log n)$ worst case time. They identify two main tasks:

- **Task A**: Keep track of the tree components during edge deletions in H_1.
- **Task B**: Keep track of the connected components during edge insertions in H_2, and the single unchecked vertex in each component.

Algorithm 2.1: 2OUTMC: Monte Carlo on 2-out graphs

Input: The multigraphs CG and RG.
Output: A perfect matching M.

```
H_1 ← CG, H_2 ← empty graph with columns as vertices
All vertices in H_1 are unmarked, all vertices in H_2 are unchecked
CORE ← edges in cycles of CG
while there exists a tree T in H_1 with no marked vertex do
    Let x be a random vertex of T{x is a column vertex in the original graph}
    marked[x] ← true {x must be matched with one of its choices}
    Add the edge of x in H_2
    Let Q_x be the component in H_2 containing the edge of x
    if Q_x has no unchecked vertices then
        Return Fail {Q_x has more edges than vertices (no 1-1 pairing possible)}
    else
        Select an unchecked vertex y of Q_x. In case of ties, prefer one from CORE
        checked[y] ← true {y will be matched with a row that selected it}
        delete y in H_1 {The algorithm forgets y's choices}
Create B'_2o from B_2o by keeping only edges between marked rows and checked columns (edges in
    H_2) or unmarked rows and unchecked columns (edges in H_1)
Apply KS_R1 on B'_2o to find the perfect matching M
```

Task B can be efficiently done in amortized near linear time (over the course of the algorithm) by using a union-find data-structure and keeping the identity of the single unchecked vertex in a component of H_2 at the root of the component. For Task A, Karp et al. propose the following. In the beginning, the edges of CG are labeled as $\mathcal{F}$, if their deletion creates a tree; $\mathcal{T}$, if they belong to a tree component; and $\mathcal{C}$ otherwise. Let c-degree of a vertex v be the number of $\mathcal{C}$ edges incident on v. During deleting the edge (u, v) from H_1, one of the following is performed depending on the label of (u, v).

- **Case 1**: (u, v) is $\mathcal{C}$: The c-degrees of u and v are decreased by one. Then, while there is a vertex with a single $\mathcal{C}$ edge; its $\mathcal{C}$ edge is relabeled as $\mathcal{F}$.
- **Case 2**: (u, v) is $\mathcal{F}$: Using a dove-tailed depth-first search, where depth-first searches from u and v are interleaved, the tree component created can be found in time proportional to its size. One then changes the labels of all edges in this tree from $\mathcal{F}$ to $\mathcal{T}$.
- **Case 3**: (u, v) is $\mathcal{T}$: Deleting (u, v) creates two trees. As in the previous case, a dove-tailed DFS is used to find these two trees in time proportional to the size of the smaller one. The new trees are to be examined by the algorithm.

We identify an oversight in this procedure, where the algorithm fails to keep track of some trees in H_1. We demonstrate this by an example. In Figure 2.7, if the edge between vertices u and v gets deleted, then the connected component is split into two triangles. The c-degree of both u and v decreases to two, and as both are greater to one, the deletion procedure stops without any action. However, both triangles are unicylic. If an edge is deleted from either triangle, then Case 1 will not recognize that the remaining edges should be relabeled as $\mathcal{T}$ instead of $\mathcal{F}$. We propose a fix for this oversight in Lemma 2.2.

Lemma 2.2. *Let u be an endpoint of a deleted edge (u, v) with label $\mathcal{C}$. Apply the procedure of Case-1 until we arrive at a vertex p with* c-degree$[p] \neq 1$. *If* c-degree$[p] = 0$, *then u's component has become a tree.*

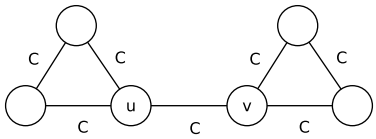


Figure 2.7 – Algorithm 2.1 does not recognize the creation of a tree if any edge is deleted after (u, v).

Proof. We claim that if c-degree$[p] = 0$, then p and v are the same vertex. Each vertex on the path from u to p had its c-degree affected twice (from 2 to 0), except p. Hence for p to become 0, its c-degree must have been equal to 1. If $p \neq v$, then p should had its $\mathcal{C}$ edge relabeled during another deletion process. Therefore, prior to the deletion of (u, v), there was a cycle on H_1 with all vertices having c-degree equal to 2, and both their $\mathcal{C}$ edges participated in the cycle. Any outgoing edges from vertices of the cycle therefore were labeled $\mathcal{F}$ and by definition, their deletion led to a tree being formed. The component was hence unicyclic before. □

Case 1-continuation is therefore as follows:

- Once there are no vertices with c-degree equal to 1, take the last vertex v whose c-degree was reduced. If c-degree$[v] = 0$, then relabel all edges in vs component from $\mathcal{F}$ to $\mathcal{T}$.

This addition has overall $\mathcal{O}(n)$ cost, because each edge can change label at most twice.

2.3.1.2 Conversion to an efficient general heuristic

Algorithm 2.1 works well when the random 2-out graph is sampled from $K_{n,n}$. However, in the case of an arbitrary host graph, the underlying theory is not shown to hold, and the algorithm can make erroneous decisions. Here we discuss how to turn Algorithm 2.1 into a general heuristic. Apart from the aim of obtaining a practical heuristic for bipartite matching, there is another reason to investigate the matching problem in 2-out bipartite graphs. We will show subsequently that an $\mathcal{O}(f(n, m))$ time algorithm to find a maximum cardinality matching in a 2-out bipartite graph can be used to find a maximum cardinality matching in any bipartite graph with m edges in $\mathcal{O}(f(m, m))$ time, where f is a function on the number of vertices n per side and edges m. Such a reduction is important because it shows that an algorithm for finding maximum cardinality matchings in 2-out graphs with similar complexity to 2OUTMC can be used to obtain an $\mathcal{O}(m \log m)$ algorithm for matchings in general bipartite graphs.

If Algorithm 2.1 reaches Line 10 during execution, it quits immediately before examining all trees in H_1. We instead propose to continue with the execution of the algorithm to make the returned matching as large as possible. To achieve this efficiently, we keep for each tree T a list L_T of unmarked vertices. At Line 5 we randomly sample x from L_T and discard it from L_T. Contrary to Algorithm 2.1, we neither mark x nor insert it in H_2 yet. Instead, we examine first whether the component in H_2 of either of the two choices of x has an unchecked column y. If y exists, we mark x, insert it to H_2 and continue by deleting y from H_1. Otherwise, we perform the same set of actions with another randomly sampled vertex from L_T. If L_T becomes empty, and no vertex was marked, we abandon T and proceed to another tree. Each such tree in the final state of H_1 decreases the cardinality of the returned matching by one, as a row is left free. If T is split into two trees, the lists of unmarked vertices for the new trees contain only those vertices still inside L_T at the moment of splitting. This is necessary to avoid sampling vertices more than once.

The overall algorithm 2OUTMC is as follows. It takes the matrix representation $\mathbf{A}$ of the given bipartite graph in the CSC/CSR format and scales it with a few steps of the Sinkhorn–Knopp algorithm to obtain the scaled matrix $\mathbf{S} = \mathbf{RAC}$. It then chooses two random neighbors for each column and row using their respective probability distributions in the corresponding rows and columns of $\mathbf{S}$, which are given as input to Algorithm 2.1. Then, the auxiliary graph B_{2o} is constructed and KS_{R1} is run on this graph to retrieve a maximum cardinality matching in B_{2o}. If one allows vertices to choose neighbors uniformly, then there are no guarantees on the maximum cardinality of a matching in B_{2o}. As an example, consider the graph where the ith row and ith column are connected for $i = 1, \ldots, n$, and additionally the first ℓ rows and columns are connected with every vertex on the opposite side. Then, in expectation $\mathcal{O}(\frac{\ell-1}{\ell+1} \cdot n)$ rows (resp. columns) make both choices from the first ℓ columns (resp. rows), such that in the generated B_{2o} the maximum cardinality matching is of size $\mathcal{O}(\frac{n}{\ell}+\ell)$. Using $\mathbf{S}$'s values to perform the random choices spreads the choices so that the maximum cardinality of the matching in the subgraph increases (see Theorem 2 and Lemmas 6–8 in [38] that examines the 1-out subgraph model).

2.3.1.3 Additional heuristics for 2OUTMC

We can improve the matching quality of 2OUTMC by considering the following two heuristics.

Heuristic 1: Delayed tree vertex selection during Line 5. The ideal case at Line 5 of Algorithm 2.1 is to select an x such that x's insertion as an edge to H_2 does not lead to a new tree in H_1 after the deletion of the edge corresponding to the unchecked vertex of the connected component Q_x. This is only possible if Q_x contains an unchecked column labeled as $\mathcal{C}$ in H_1. Otherwise, a new tree will be created in H_1, and the algorithm will have to process it in a future step.

For the first heuristic, we greedily select an x such that, if possible, the creation of a tree in H_1 is avoided. We replace L_T is with two lists L_T^1 and L_T^2. The lists L_T^1 contains those unmarked vertices of T whose insertion in H_2 leads to a new tree; L_T^2 contains all other L_T vertices that have not been tried yet. At first, we sample x from L_T^2 and see whether the components of x's choices in H_2 have an unchecked vertex of type $\mathcal{C}$ in H_1. If they have, x is marked and inserted to H_2. Otherwise, x is inserted in L_T^1, and we consider another random vertex of L_T^2. If L_T^2 becomes empty, we start sampling from L_T^1. With the union-find data structure, this heuristic requires constant amortized time per sample and each vertex can be sampled at most twice. Therefore the overhead associated with this heuristic is almost linear in n.

Heuristic 2: Online creation of the RG multigraph. In this heuristic, the decisions of the rows are not given as input, but are instead defined during the course of the algorithm. Similar to the previous idea, this heuristic aims to reduce the possibility that a tree in H_1 gets created following an edge insertion into H_2.

More specifically, consider a vertex x randomly chosen at Line 5. In this heuristic, x has not picked its two choices yet, and we let x choose them at this point, in the way that benefits the algorithm the most. This is done as follows. Initially, we iterate over all of x's neighbors in the host graph G. Let c be one of x's neighbors and c^* be the sole unchecked vertex in c's connected component in H_2, or $c^* = -1$ if no unchecked vertices exist. We assign values to x's neighbors to classify them. If c^* is equal to -1, c's value is 0. If c^* has label $\mathcal{F}$ or $\mathcal{T}$ in H_1, c's value is 1. Otherwise, c's value is 2. Based on these assigned values, we partition the neighbors of x in G

into three disjoint sets C_0, C_1 and C_2 such that C_i contains all neighbors of x with value equal to i. Selecting columns from C_2 is preferred, as they can avoid creating a tree in H_1. Vertex x will attempt to sample first from C_2, and if needed from C_1 or C_0, with a preference for C_1 over C_0. The sets C_0, C_1 and C_2 are kept implicitly, and each vertex x requires amortized $\mathcal{O}(d_x)$ to make its choices, where d_x is its degree. Hence, the overhead associated with this heuristic is almost linear in m.

2.3.1.4 Reducing bipartite graph matching to matching on 2-out graphs

Here, we will show that an $\mathcal{O}(f(n,m))$ time algorithm to find a maximum cardinality matching in a 2-out bipartite graph can be used to find a maximum cardinality matching in any bipartite graph with m edges in $\mathcal{O}(f(m,m))$ time, where f is a function on the number of vertices n and edges m. We have already discussed the complications of this result in the previous section.

Let $G = (V_G, E_G)$, with be a graph with minimum degree at least two. If G's minimum degree is one, we can apply Rule-1 of Karp–Sipser to match degree-1 vertices with their neighbors and consider as G the resulting graph.

We produce a new graph G' from G in the following way. For any edge $e = (a, b) \in E$ we add edges $e' = (a, a_e)$, $e'' = (a_e, b_e)$, and $e''' = (b_e, b)$ to G'. We hence introduce two new vertices a_e, b_e s.t $d_{G'}(a_e) = d_{G'} = 2$ for each edge $e \in E_G$. The degree of nodes in V_G remains unchanged in G'.

Lemma 2.3. *Let H be a random 2-out subgraph G'. Then $H = G'$.*

Proof. The added vertices a_e, b_e have degree two and will select both neighbors, hence no edge will remain unpicked. □

In what follows, we refer to the second reduction rule of Karp–Sipser which merges the neighbors of a degree-2 vertex, which is then discarded, as a degree-2 reduction.

Lemma 2.4. *It is possible to obtain G by doing only degree-2 reductions on G'.*

Proof. Let a_e be a vertex of G', introduced due to the edge $e = (a, b)$. Since $d_{G'}(a_e) = 2$ we can apply a degree-2 reduction which will merge a with b_e to create a single node ab_e. As a consequence of this merge, the edge (ab_e, b) will be created and edges $(a, a_e), (a_e, b_e), (b_e, b)$ will be erased. We simply relabel ab_e to a again. The proof then follows similarly by applying degree-2 reduction for all a_e corresponding to $e \in E$ until we obtain G. □

Now we show that maximum matchings in G' are related to those on G and vice versa.

Lemma 2.5. *Any maximum cardinality matching M' on G' corresponds to a maximum cardinality matching M on G.*

Proof. Let M' be a maximum cardinality matching on G'. A matching M for G can be generated in the following way: If both (a, a_e) and (b_e, b) appear in M', e is added to M. Hence it suffices to show that any maximum cardinality matching M' in G' necessarily contains $|M|$ pair of matched edges (a, a_e) and (b, b_e).

First, we have that $|M'| = |E_G| + |M|$. To see this, note that per Lemma 2.4 we perform $|E_G|$ degree-2 reductions, and result in G. Each of this reductions corresponds with a matched edge in M'. Then, we only need to find the maximum cardinality on G which is $|M|$.

Let S_a contain all indices e such that (a, a_e) is in M' and (b_e, b) is not in M'. Set S_b is defined similarly. Set $S_\emptyset$ contains all indices e such that (a_e, b_e) appears in M'. Finally, S_{ab}

contains all indices e such that (a, a_e) and (b, b_e) are matched together in M'. Then, since M' is a maximum cardinality matching we have

$$|S_a| + |S_b| + |S_\emptyset| + 2 \cdot |S_{ab}| = |E_G| + |M| \ .$$

This is true because of the fact that for each edge e exactly one matched edge appears in M' in case $e \in S_a \cup S_b \cup S_\emptyset$ and two edges are added if $e \in S_{ab}$.

However, $|S_a| + |S_b| + |S_\emptyset| + |S_{ab}| = |E_G|$, since each edge e must appear in one of those sets and there exist exactly $|E_G|$ of them.

Hence, $|S_{ab}| = |M|$ necessarily. As they define a matching in G and their cardinality is $|M|$, the matching is maximum. □

Using the above lemma, we can prove Theorem 2.4 below.

Theorem 2.4. *Assume there is an algorithm* `ALG` *working in* $\mathcal{O}(f(n, m))$ *time for finding a maximum cardinality matching in a 2-out graph. Then we can find a maximum cardinality matching in* $\mathcal{O}(f(m, m))$ *time for any given graph.*

Proof. Let G be any bipartite graph without degree-1 vertices and $m = |E_G|$. In $\mathcal{O}(m)$ time we generate G'. By Lemma 2.3, the 2-out subgraph of G' corresponds to G' itself. In addition $|E_{G'}|, |V_{G'}| \in \mathcal{O}(m)$. Using `ALG`, we can find a maximum cardinality M' for G' in $\mathcal{O}(f(m, m))$ time. By Lemma 2.5 then, we can convert M' to a maximum cardinality matching M for G in $\mathcal{O}(m)$ time. □

As a byproduct of Lemma 2.5, we observe that the transformation of G to G' also eliminates the need to apply SK as a preprocessing step.

2.3.2 TruncRW: Truncated random walk with nonuniform sampling

2.3.2.1 Description of the algorithm for regular bipartite graphs

Goel et al. [54] propose a randomized algorithm (of the Las Vegas type) that finds a perfect matching in a d-regular bipartite graph with n vertices in each side in $\mathcal{O}(n \log n)$ time in expectation. This algorithm starts a random walk from a randomly chosen free column-vertex. At a column vertex c, the algorithm selects uniformly at random one of the row-vertices that are not matched to c, and goes to the chosen row vertex r. If r is free, then an augmenting path is obtained by removing possible loops from the walk. If r is matched, then the random walk goes to the mate of r. Goel et al. show that the total length of the random walks is $\mathcal{O}(n \log n)$ in expectation, and thus the algorithm obtains a perfect matching in the stated time [54, Theorem 4]. They also show that one can obtain a Monte Carlo-type algorithm by truncating the random walks. The expected length of an augmenting path with respect to a given matching of cardinality j is $2(4 + 2n/(n - j))$, and the random walks could be truncated at this length to obtain near optimal matchings in $\mathcal{O}(n \log n)$ time.

A random walk is easy to implement for d-regular bipartite graphs. At a column vertex c, one can create a random number between 1 and d in $\mathcal{O}(1)$ time and choose the neighbor at that position, and repeat the experiment if the mate of c is chosen. This will take $\mathcal{O}(1)$ time in expectation for each step of the walk, and the run time bound of $\mathcal{O}(n \log n)$ is maintained.

Goel et al. show that the random-walk based algorithm will work for finding perfect matchings in the bipartite graph representation of a doubly stochastic matrix. When a given matrix has constant row and column sums and each entry is integral (in which case the matrix is doubly

$$\begin{pmatrix} \sqrt{2} & & & \\ & \frac{1}{\sqrt{2}} & & \\ & & \frac{1}{\sqrt{8}} & \\ & & & \frac{1}{\sqrt{8}} \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{8}} & & & \\ & \frac{1}{\sqrt{8}} & & \\ & & \frac{1}{\sqrt{2}} & \\ & & & \sqrt{2} \end{pmatrix} = \begin{pmatrix} 1/2 & 1/2 & 0 & 0 \\ 1/4 & 1/4 & 1/2 & 0 \\ 1/8 & 1/8 & 1/4 & 1/2 \\ 1/8 & 1/8 & 1/4 & 1/2 \end{pmatrix}$$

Figure 2.8 – The matrix $\mathbf{A}$ associated with a 4×4 Hessenberg matrix, the scaling matrices $\mathbf{R}$ and $\mathbf{C}$, and the resulting doubly stochastic matrix $\mathbf{S} = \mathbf{RAC}$. In general, $\mathbf{S}(n, 1) = 1/2^{n-1}$.

stochastic), by using an existing data structure [57] one can attain the same $\mathcal{O}(n \log n)$ run time bound. For general doubly stochastic matrices without any bound on the entries, Goel et al. propose an augmented binary search tree with which each selection step of the random walk can be implemented in $\mathcal{O}(\log n)$ time, and obtain a run time of $\mathcal{O}(m + n \log^2 n)$ in expectation, with a total of $\mathcal{O}(m)$ preprocessing time.

2.3.2.2 Conversion to an efficient general heuristic

Let c be a free column vertex with respect to a given matching of cardinality j. Assuming there is a perfect matching, one can find an augmenting path to match c, and a random walk can find it. The $\mathcal{O}(\frac{n}{n-j})$ bound on the expected length of such a path will not hold if the bipartite graph is not regular. One may perform more than m steps, which is the worst case time complexity of deterministically finding an augmenting path starting from a free vertex. We propose two methods to make the random walks more useful and to sample efficiently in a random walk. We also discuss an efficient implementation of the whole approach.

The first proposed method is to scale the matrix representation $\mathbf{A}$ of a given bipartite graph to obtain a doubly stochastic matrix $\mathbf{S} = \mathbf{RAC}$ for random selections. The expected length of a random walk to find an augmenting path holds when $\mathbf{S}$ has bounded nonzero entries. In general, ones does not have any bound on the entries of $\mathbf{S}$. Consider the matrix $\mathbf{A}$ associated with an upper Hessenberg matrix of size n. $\mathbf{A}$ has a full lower triangular part, and additional $n - 1$ entries $\mathbf{A}(i - 1, i) = 1$ for $i = 2, \ldots, n$, and fully indecomposable. The 4×4 example along with its unique scaling matrices are shown in Figure 2.8. In the resulting scaled matrix $\mathbf{S}(n, 1) = 1/2^{n-1}$ whose inverse is not bounded polynomially in n.

As highlighted in the above paragraph, one needs an $\mathcal{O}(\log n)$ time algorithm to select a row vertex randomly from a given column vertex. The second proposed method is a simple yet efficient algorithm for this purpose, rather than a sophisticated augmented tree. The main components of the proposed sampling method are as follows. For each column vertex c, with d_c neighbors, we have:

- $\mathsf{adj}_c[1, \ldots, d_c]$: an array keeping the neighbors of c.
- $\mathsf{wghts}_c[1, \ldots, d_c]$: the weight of the edges incident on c. This array is parallel to the first one so that the weight of the edge $(c, \mathsf{adj}_c[i])$ is $\mathsf{wghts}_c[i]$.
- $\mathsf{medge}[c]$: the position of the mate of c in the array adj_c, or -1 if c is not matched.

At the beginning, we compute the prefix sum of $\mathsf{wghts}_c[1, \ldots, d_c]$. After this operation, the total weight of the edges incident on c is $\mathsf{wghts}_c[d_c]$, and the weight of the edge $(c, \mathsf{mate}[c])$ is $\mathsf{wghts}_c[\mathsf{medge}[c]] - \mathsf{wghts}_c[\mathsf{medge}[c] - 1]$, assuming that $\mathsf{wghts}_c[0]$ signifies zero.

Given the prefix sums in $\mathsf{wghts}_c[1, \ldots, d_c]$, the position of the mate of c at $\mathsf{medge}[c]$, we can choose a random neighbor (which is not equal to $\mathsf{mate}[c]$) as shown in Algorithm 2.2. We use a binary search function, binSearch, which takes an array, the array's start and end positions, a

target value, and returns the smallest index of an array element which is larger than the given value with binary search (we skip the details of this search function). At Line 5, since c does not have a mate, we search in the whole list. At Line 8, since the prefix sum just before $\mathsf{medge}[c]$ is larger than the target value, we search in the first part of wghts_c until the current mate located at $\mathsf{medge}[c]$. At Line 10, we search on the right of $\mathsf{medge}[c]$, by a modified target value. This last part is the gist of the algorithm's efficiency as it avoids updating the prefix sums when the mate changes.

Algorithm 2.2: SAMPLE: Algorithm to sample a random neighbor of a column vertex c with d_c neighbors

Input: $\mathsf{adj}_c[1,\ldots,d_c]$, $\mathsf{wghts}_c[1,\ldots,d_c]$, and $\mathsf{medge}[c]$.
Output: A random neighbor of column c.

1: mwght $\leftarrow \mathsf{wghts}_c[\mathsf{medge}[c]] - \mathsf{wghts}_c[\mathsf{medge}[c]-1]$ if $\mathsf{medge}[c] \neq -1$, otherwise 0
2: totalW $\leftarrow \mathsf{wghts}_c[d_c] -$ mwght{The total weight of the edges that can be sampled}
3: create a random value rv between 0 and totalW
4: **if** $\mathsf{medge}_c = -1$ **then**
5: **return** binarySearch($\mathsf{wghts}_c[1,\ldots,d_c], rv$)
6: **else**
7: **if** $\mathsf{wghts}_c[\mathsf{medge}_c]-$mwght $\geq rv$ **then**
8: **return** binSearch($\mathsf{wghts}_c[1,\ldots,\mathsf{medge}[c]-1], rv$)
9: **else**
10: **return** binSearch($\mathsf{wghts}_c[\mathsf{medge}[c]+1,\ldots,d_c], rv +$ mwght$) + \mathsf{medge}_c$

The sampling algorithm returns the index of the neighbor in adj_c different from the current mate in time $\mathcal{O}(\log d_c)$, independent of the values of the edges. It thus respects the required run time bound. If we were to apply the rejection sampling (as discussed before for the regular bipartite graphs), the run time would depend on the value of the matching edge that we want to avoid. This could of course lead to an expected run time of more than $\mathcal{O}(n)$.

There are two key components of Algorithm 2.2. The first one is the prefix sum, which is computed once before the random walks start and does not change. The second one is $\mathsf{medge}[c]$, the position of $\mathsf{mate}[c]$ in adj_c. The value $\mathsf{medge}[c]$ changes and needs to be updated when we perform an augmentation. We handle this update as follows. We keep the random walk in a stack by storing only the column vertices, as the row vertices direct the walk to their mate, or terminate the walk if not matched. We discard the cycles from the random walk as soon as they arise—this way we only store a path on the stack, and its length can be at most n. Storing a path also enables keeping the $\mathsf{medge}[\cdot]$ up-to-date. Every time we sample an outgoing edge from a column vertex c, we assign the location of the sampled row vertex in adj_c to a variable $\mathsf{nmedge}[c]$. When we find a free row, the stack contains the column vertices of the corresponding augmenting path, whose new mates' locations are in $\mathsf{nmedge}[\cdot]$ and thus can be used to update $\mathsf{medge}[\cdot]$.

The described procedure will work gracefully in expected $\mathcal{O}(m + n \log n)$ time for regular bipartite graphs and for doubly stochastic matrices where the nonzero values do not differ by large. On the other hand, when there are large differences in edge weights, a random walk can get stuck in a cycle. That is why truncating the long walks is necessary to make the algorithm work for any given doubly stochastic matrix. Furthermore, such a truncation is necessary with the proposed matrix scaling approach for defining random choices. For the overall approach to be practical, we should not apply the scaling algorithms until convergence. As in the previous approaches [38], we allot a linear time of $\mathcal{O}(m + n)$ for scaling. Applying Sinkhorn–Knopp

algorithm for a few iterations will thus be allowable. The known convergence bounds for the Sinkhorn–Knopp algorithm [78, Theorem. 4.5] apply asymptotically, therefore we do not have any bounds on the error after a few iterations; it can be large. That is why truncation makes the random walk based augmenting path search practical.

The overall algorithm TruncRW is thus as follows. It takes the matrix representation of the given bipartite graph and scales it with a few steps of the Sinkhorn–Knopp algorithm. Then for $j = 0$ to $n-1$, it uniformly at random picks a free column vertex, and starts a random walk starting from that column, for at most $2(4+2n/(n-j))$ steps, after which the walk is truncated.

2.3.2.3 Further comments on TruncRW

We incorporated a known heuristic called look-ahead [33, 35] for speeding up the augmenting path search in practice. In this heuristic, before sampling an arbitrary row-vertex from a column-vertex c, we check if there is a free row vertex in the adjacency list of c. If so, such a row is returned, and the random walk terminates. The implementation of this heuristic has a total overhead of $\mathcal{O}(m)$ for the whole course of the algorithm [33, 35]. We note that the look-ahead technique trades the quality of TruncRW with run time. In our experiments, the look-ahead heuristic reduced the run time significantly; it interferes with the randomization though.

We can easily apply TruncRW to bipartite graphs with different number of vertices in each side. This is based on the fact that we can scale a rectangular $n_r \times n_c$ matrix (say $n_r \geq n_c$) so that all columns have sum of 1, and all rows have equal sum of n_c/n_r, if there is matching covering all columns, and all entries can be put in such a matching. Then, all components of TruncRW work without any change.

If there is no total support, then Sinkhorn–Knopp works in such a way that the entries that cannot put into a perfect matching tend to zero. This is helpful in TruncRW's context, as the corresponding edges will not likely be selected in a random walk. If there is no perfect matching, then little is known about scaling. It is our experience that the Sinkhorn–Knopp iterations tend to zero out entries that cannot be put into a maximum cardinality matching. Therefore, in this case again, scaling, random selection, and truncation should help. We present some experiments later to support this observation and leave the question of showing this theoretically as an open problem.

2.3.3 Experiments

We implemented 2outMC and TruncRW in C/C++. The codes, all are sequential, were compiled with "-O3" and run on a machine with 2 x Intel Xeon CPU Gold 6136 CPUs and 187 GB RAM. Throughout this Section we use a set of 39 large sparse square matrices from the SuiteSparse Matrix Collection [31]. These matrices are automatically selected from all square matrices available at the collection with $10^6 \leq n \leq 28 \times 10^6$, and with at least two nonzeros per row or column. All of the bipartite graphs have perfect matchings. Apart from these real-life graphs, we also evaluate 2outMC and TruncRW on some synthetic bipartite graphs with equal number of vertices in each side. We compared our two heuristics against KS, and KS_{R1}. We use our implementation of KS from the previous part, but opted for an array-based implementation for KS_{R1}. In addition, we investigated if random 2-out and 3-out bipartite graphs of a general host graph have perfect matchings if rows and columns select neighbors with the probabilities in the scaled matrix representation. The practical version of Sinkhorn–Knopp which only scales a matrix for a few iterations is referred to as SK-t, where t is the number of allowed iterations. All run times are reported in seconds.

$\frac{m}{n}$	[0,10)		[10,20)		[20,30)		[30,40)		[40,50)	
#Instances	27		5		5		1		1	
	#PM	deficiency	#PM	deficiency	#PM	deficiency	#PM	deficiency	#PM	deficiency
2-out Model M_1	0	223	0	8	1	20	0	2	1	0
2-out Model M_2	27	0	3	3	1	10	0	1	0	1
3-out Model M_1	27	0	5	0	5	0	1	0	1	0
3-out Model M_2	27	0	5	0	5	0	1	0	1	0

Table 2.4 – We divide the real-life graphs into five groups. The ith group consists of graphs whose $\frac{m}{n}$ ratio is between $10(i-1)$ and $10i$. For each group, we give the number of instances in which 2-out and 3-out graphs built using the models M_1 and M_2 have a perfect matching and the largest difference from the maximum cardinality of a matching.

2.3.3.1 Investigation of perfect matchings in k-out graphs

Here, we investigate the claim that 2-out and 3-out random graphs will likely have a perfect matching, if created with the probabilities in the scaled matrix. For this test, we used the 39 real-life graphs mentioned at the beginning of the section.

We consider two different models to create a random 2-out graph. In the model M_1, row choices are independent of the column choices. Under this model, a row and a column can select each other resulting in parallel edges—only one of them is kept. The model M_2 tries to avoid parallel edges. In this model, all columns perform their selections. Then, each row r attempts to randomly choose two columns, only from those that did not select r. These selections again are based on the scaled matrix. In this model, parallel edges can arise (and be discarded) only when a vertex v is connected in the 2-out graph with all of its neighbors in G, because it is impossible for v to select otherwise. These models naturally extend themselves to 3-out graphs as well.

We experimented three times with each real-life graph. M_i's result is the maximum of those three experiments. In each test, we first created the choices of all columns. Then we allowed the two models to generate the choices of the rows accordingly.

The results are shown in Table 2.4 for the 39 real-life graphs and are with SK-5. As seen in this table, the random 2-graphs generated with the model M_1 have near perfect matchings, but they do not contain perfect matchings in most cases. In contrast, the random 2-graphs generated by M_2 in many cases contain a perfect matching. In only a few graphs this does not hold true, and in these cases the deficiency is no more than 10.

Interestingly, when we pass from $k = 2$ to $k = 3$ we see that both models M_1 and M_2 were able to find a perfect matching in all 39 graphs. Including $\mathcal{O}(n)$ extra edges in the 2-out graphs sufficed to make them have a perfect matching.

2.3.3.2 On synthetic graphs

In Table 2.5, we give results with a synthetic family $\mathcal{J}$ of graphs from literature [38], whose matrix representations do not have total support. To create a member of $\mathcal{J}$, we separate the vertex set R into $R_1 = \{r_1, \ldots, r_{n/2}\}$ and $R_2 = \{r_{n/2+1}, \ldots, r_n\}$ and likewise for C. All vertices of R_1 are connected to all vertices of C_1. Edges $(r_i, c_{n/2+i})$ and $(r_{n/2+i}, c_i)$ for $i = 1, \ldots, n/2$ are added to introduce a perfect matching. A parameter t is used to connect t vertices from R_1, and t vertices from C_1 to every vertex on the opposite side.

As seen in Table 2.5, both KS and KS_{R1} have more and more difficulty with increasing t. The matching quality drops over 30% between $t = 2$ and $t = 512$ for KS and almost 40% for KS_{R1}.

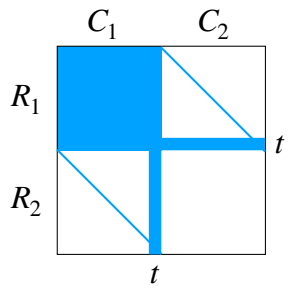


Figure 2.9 – $\mathbf{A}_{\mathcal{J}}$: Adjacency matrix representation for the synthetic family $\mathcal{J}$.

			2OUTMC		TRUNCRW	
t	KS_{R1}	KS	Uniform	SK-5	Uniform	SK-5
2	0.93	1.00	0.78	0.99	0.88	0.99
8	0.80	0.85	0.59	0.99	0.91	0.99
32	0.69	0.72	0.52	0.99	0.83	0.99
128	0.64	0.65	0.51	0.99	0.78	0.99
512	0.61	0.63	0.52	0.99	0.76	0.99

Table 2.5 – Average quality of the matchings found by the algorithms on graphs from the synthetic family $\mathcal{J}$ for $n = 30000$ and various values of t.

On the contrary, 2OUTMC and TRUNCRW both obtain a near perfect matching, with SK-5. Even though the matrices associated with the graphs of $\mathcal{I}$ lack total support, SK-5 sufficed to obtain near optimal matchings. We notice the effect of scaling: if vertices select without scaling (Uniform), the matching quality reduces. This is particularly true for 2OUTMC, which exhibits the worst overall performance with uniform selection. Family $\mathcal{J}$ shows the importance of scaling, and more importantly highlights the robustness of the proposed methods. An adversary can create graphs which make degree-based randomized approaches lose quality—some of those heuristics were described in Section 2.1, and the full details including negative results on KS_{R1} can be found elsewhere [12]. On the other hand, the use of scaling helps to avoid such cases for 2OUTMC and TRUNCRW.

We now discuss how to slightly modify the synthetic family $\mathcal{I}$ seen in the experiments of Section 2.2.5 to obtain a new synthetic family $\mathcal{I}'$ for which both KS, and KS_{R1} have reduced performance. Recall that the adjacency matrix of the graphs in the $\mathcal{I}$ family consisted of an upper triangular matrix enhanced by two extra nonzero entries $(2, 1)$ and $(n, n\text{-}1)$. There was thus a set of four degree-2 vertices $\{r_n, r_{n-1}, c_1, c_2\}$, where executing Rule-2 on any of them lead to a perfect matching. To create $\mathcal{I}'$, we extend $\mathcal{I}$ with the following edges (r_3, c_1), (r_3, c_2), (r_n, c_{n-2}) and (r_{n-1}, c_{n-2}). One can then see that the lowest degree is 3 and thus both both KS, and KS_{R1} will resort to random suboptimal decisions. Likewise, due to the large number of entries without support in the matrix representation, Sinkhorn–Knopp will take many iterations to properly scale the matrix. In Table 2.6, we give results of the algorithms for a few graphs from the family $\mathcal{I}'$. In the table, we also show the effects of scaling on 2OUTMC and TRUNCRW by showing results without scaling (under column "uniform", in which a column vertex chooses a neighbor uniformly at random), with SK-5, and with SK-20. As can be seen, despite the lack of total support, both 2OUTMC and TRUNCRW obtain matchings whose cardinality is more than 0.92 of the maximum, when SK-5 or SK-20 is used. TRUNCRW in particular is nearly optimal. These results are always better than that of KS_{R1} and KS, with the difference in matching

	KS_{R1}	KS	2OUTMC			TRUNCRW		
n	quality	quality	uniform	SK-5	SK-20	uniform	SK-5	SK-20
10000	0.76	0.84	0.81	0.92	0.95	0.97	0.97	0.97
20000	0.73	0.83	0.81	0.92	0.95	0.97	0.97	0.97
30000	0.73	0.83	0.81	0.92	0.95	0.97	0.97	0.97

Table 2.6 – Average quality of the matchings found by the algorithms on graphs from the synthetic family $\mathcal{J}'$ for $n \in \{10000, 20000, 30000\}$.

quality being about 20–25% for the former, and 10–15% for the latter. With increased iterations of Sinkhorn–Knopp, 2OUTMC increases the cardinality of its matchings by 3%. If we do not use scaling ("uniform"), while there's no noticeable effect on TRUNCRW's matchings, 2OUTMC matchings decrease by roughly 10%. Even so, its results remain better than KS_{R1}'s and on par with those of KS.

2.3.3.3 On real-life graphs

We compared TRUNCRW and 2OUTMC with KS_{R1} and KS on the 39 real-life graphs mentioned at the beginning of Section 2.3.3. Figure 2.10(a) and Figure 2.10(b) present the high level picture. For the experiments, we did not permute the matrices randomly, which generally increases the experimentation time.

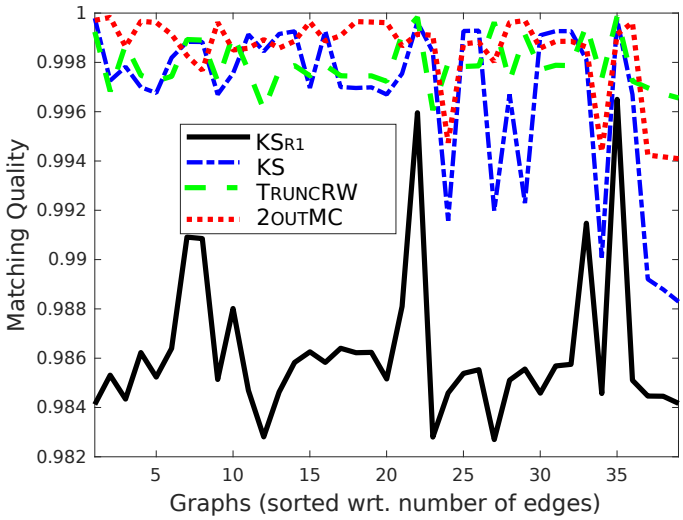
The results for matching quality can be seen in Figure 2.10(a), where we plot the ratio of the cardinality of the matchings found by different algorithms to the maximum cardinality of the matching. The graphs are indexed in nondecreasing number of edges. 2OUTMC and TRUNCRW use SK-3 for scaling. As can be observed, both 2OUTMC and TRUNCRW obtain near perfect matchings. The average matching quality obtained by 2OUTMC is 0.9979 and that obtained by TRUNCRW is 0.9984. Both algorithms never drop below 0.9900 in any of the 39 cases.

Figure 2.10(a) also shows the matching quality of KS and KS_{R1}. The good behavior of both KS and KS_{R1} on real-life instances was already discussed in the previous experiments in Section 2.2.6 and holds for these graphs as well. KS_{R1} obtains matchings of quality 0.9862 on average, with always smaller cardinality than TRUNCRW and 2OUTMC. KS fares better and its average quality is 0.9968. Even so, in the majority of cases, it obtains matchings that are inferior quality-wise to both TRUNCRW and 2OUTMC.

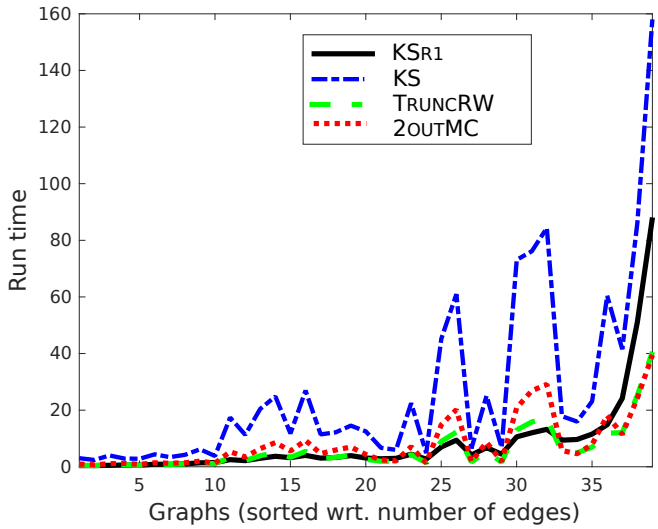
While all algorithms obtain matchings of high quality, the absolute different is remarkable in some cases. For example, the largest difference observed between the matching cardinalities obtained by 2OUTMC and KS was 346577, in favor of 2OUTMC.

Figure 2.10(b) shows the run time of all examined heuristics, where the graphs are again indexed in nondecreasing number of edges. KS_{R1} is in general the fastest of these four algorithms when there are not too many edges. TRUNCRW and 2OUTMC are close run-time wise to KS_{R1} and in some instances faster than it. This is especially true in instances with many edges because KS_{R1} depends more on the number of edges, m, especially at the initial randomization step. KS has the slowest performance overall. Nonetheless, for the most part the difference between KS and KS_{R1} is comparable with the results in Section 2.2.5.

For a detailed study, we show results on the five largest graphs from the mentioned dataset and `Circuit5M`, which was identified as a challenging instance in earlier work [76]. Degree-1 vertices from `Circuit5M` are removed by applying Rule-1 of KS as a preprocessing step—this is without loss of generality of the heuristics. For each graph we relabeled its row-vertices randomly and executed five tests with each algorithm.



(a) The matching quality of 2OutMC and TruncRW in comparison with KS and KS_{R1}.



(b) The run time of 2OutMC and TruncRW in comparison with KS and KS_{R1}.

Figure 2.10 – Run time and quality results of all algorithms on the full set of the 39 real-life graphs from Section 2.3.3. The results are with SK-3. Graphs are indexed in nondecreasing number of edges.

Table 2.7 shows the matching quality and the run time of the four heuristics. 2OUTMC and TRUNCRW used SK-3 for this set of experiments. For each graph, we give the minimum, maximum, and averages over five runs. As already discussed, all heuristics obtain high quality matchings. On a closer look, we see that TRUNCRW, on average, matched 158410 more edges than KS_{R1}, and 50847 more edges than KS. Similarly 2OUTMC matched 139220 more edges than KS_{R1} on average, and 31652 more edges than KS. Interestingly, on graph `Channel-500` TRUNCRW was able to find the maximum matching.

Concerning run time, as KS_{R1} is a linear time heuristic it is expected to be the fastest. Surprisingly, TRUNCRW even with the scaling time added is faster than KS_{R1} in three instances. This is due to the fact that each iteration of the scaling algorithm takes linear time with small constants. As an algorithm on its own (without scaling time), TRUNCRW becomes the fastest one, thanks to its run time not depending on m after the initialization. 2OUTMC, though slower, also exhibits good behavior, except in `nlpkkt240`. KS has the worst run time overall. Its initialization takes more time, and its implementation is more involved. SK-3 is fast except for `nlpkkt240` where it requires about 30 seconds. The reason that SK-3 requires 30 seconds for this particular graph is due to the random permutation of its rows, which is not cache-friendly (if SK-3 is run on `nlpkkt240` using the initial ordering of rows, it finishes in less than 10 seconds). In the other cases and despite the large size of the graphs, scaling finishes in less than seven seconds. Table 2.7 additionally shows that TRUNCRW and 2OUTMC's run time performance does not seem to be affected by their random decisions. The largest difference between the result of the minimum, and the maximum run is no more than two seconds for both of these algorithms.

Combined with the results in the previous section, we conclude thus that (i) 2OUTMC and TRUNCRW always obtain near perfect matchings, while KS_{R1} and KS are not as robust; (ii) 2OUTMC and TRUNCRW are nearly as fast as the linear time algorithm KS_{R1}, and are much faster than KS.

Next, we consider the impact of our heuristics as initialization to an exact algorithm for finding a maximum cardinality matching. We first run the heuristics to obtain an initial matching, then call an exact algorithm to augment the initial matchings for maximum cardinality. Apart from the two exact algorithms PR and PF+ used earlier, here we also consider MC21 [33] from `mmaker` [35, 76] for the augmentation steps. This algorithm visits free vertices one by one and tries to match the visited vertex with a depth-first search, and hence is closely related to TRUNCRW. In this setting, differences among the qualities of initial matchings should be observable while computing an exact matching.

The statistics of five runs with MC21 are given in Table 2.8. In this table, the time spent in augmentations is given in column "augment.". The overall time to compute a maximum cardinality matching is given in column "overall', which includes the time spent in heuristics—in case of 2OUTMC and TRUNCRW it includes the scaling time as well. The runs on `nlpkkt240` did not finish within an hour and are not presented. As seen in the table, the overall time to obtain a maximum cardinality matching is always the smallest with TRUNCRW initialization. 2OUTMC is usually competitive with the faster of KS and KS_{R1}, without a clear winner. It is also interesting to note that in all graphs the worst behavior of TRUNCRW is better than the best behavior of KS and KS_{R1} and in some cases (see `cage15` or `channel-500`) significantly so. The same is almost true for 2OUTMC as well except for graphs `Delaunay_24` and `Hugebbuble-0020` where 2OUTMC's worst result is only a few seconds slower than KS_{R1}'s best result, or `cage15` versus KS.

In Table 2.9, we observe the behavior of the heuristics when used for initializing the PF+

			KS_{R1}		KS		SK-3	2OUTMC		TRUNCRW	
name	n	statistics	quality	time	quality	time	time	quality	time	quality	time
		min.	0.99	12.67	0.99	26.89	4.59	0.99	8.82	0.99	8.27
`cage15`	5.15	avg.	0.99	12.81	0.99	27.08	4.68	0.99	8.88	0.99	9.32
		max.	0.99	13.17	0.99	27.27	4.83	0.99	8.96	0.99	10.23
		min.	0.99	10.12	0.99	20.63	2.74	0.99	7.63	1.00	3.86
`Channel-500`	4.80	avg.	0.99	10.16	0.99	20.94	2.75	0.99	7.66	1.00	4.48
		max.	0.99	10.18	0.99	21.87	2.75	0.99	7.70	1.00	5.11
		min.	0.99	6.57	0.99	24.74	2.45	0.99	4.40	0.99	2.07
`Circuit5M`	5.55	avg.	0.99	6.76	0.99	24.93	2.84	0.99	4.56	0.99	2.19
		max.	0.99	7.03	0.99	25.33	4.16	0.99	4.81	0.99	2.35
		min.	0.99	11.58	0.99	65.97	4.32	0.99	23.34	0.99	11.21
`Delaunay_24`	16.00	avg.	0.99	11.61	0.99	68.30	4.44	0.99	23.58	0.99	11.31
		max.	0.99	11.66	0.99	72.47	4.48	0.99	24.38	0.99	11.37
		min.	0.99	14.97	0.99	91.42	6.26	0.99	30.96	0.99	14.25
`Hugebubbles-0020`	21.19	avg.	0.99	15.04	0.99	97.77	6.29	0.99	31.28	0.99	14.38
		max.	0.99	15.15	0.99	106.78	6.31	0.99	31.59	0.99	14.57
		min.	0.98	98.58	0.99	182.08	29.77	0.99	52.34	0.99	34.34
`nlpkkt240`	27.99	avg.	0.98	98.66	0.99	183.10	29.92	0.99	52.53	0.99	34.50
		max.	0.98	98.76	0.99	186.08	30.27	0.99	52.76	0.99	34.70

Table 2.7 – Full run time comparisons with heuristics on the 39 real-life graphs from Section 2.3.3. The run time of SK-3 should be added to TRUNCRW and 2OUTMC. For each instance we give the minimum, the average, and the maximum of five runs for all columns regarding the quality and the run time. The number of vertices n per side is in the order of millions.

algorithm. The table shows the minimum, average, and maximum time over the five runs. As can be observed, TRUNCRW overall exhibits the best behavior. TRUNCRW has the fastest performance in four out of six instances, and in the remaining two instances it is very close to KS_{R1}. The largest difference between the two can be observed in graph `nlpkkt240` where KS_{R1} is overall almost 50 seconds slower. The total run time with KS is never better than that with TRUNCRW. It roughly takes the same amount of time for PF+ to augment 2OUTMC's initial matching, as it takes for it to augment the matching of TRUNCRW. Therefore, when 2OUTMC has a run time similar to TRUNCRW their overall run times are similar. In the largest of instances 2OUTMC's and TRUNCRW's performance diverge, but 2OUTMC's overall behavior is superior to KS and competitive with that of KS_{R1}.

In Table 2.10, we observe the behavior of the heuristics when used for initializing the PR algorithm. The behavior of KS_{R1} in graph `Circuit5M` demonstrates the robustness of our approaches. The average behavior of PR initialized with KS_{R1} is 339 seconds with the maximum run time exceeding 500 seconds. In stark contrast, PR with TRUNCRW's input never needs more than 25 seconds, whereas with 2OUTMC it never surpasses 150 seconds. In the remaining graphs, the algorithms are competitive with KS_{R1} or even faster in some instances.

In summary, the effects of the proposed methods as an initialization routine are more observable with MC21 on all instances. With PF+, we see that the augmentations take less time on average with 2OUTMC and TRUNCRW, but the overall time with KS_{R1} can be sometimes better than that of TRUNCRW slightly thanks to KS_{R1} being faster. When PR is used, the augmentations take less time with KS_{R1} in three instances compared to TRUNCRW; and in four instances compared to 2OUTMC. When 2OUTMC and TRUNCRW serve better than KS_{R1} as an initialization to PR, the difference is more significant. The above results with three different algorithms demonstrate the merits of the two proposed algorithms for use as initialization

name	statistics	KS_{R1}		KS		2OUTMC		TRUNCRW	
		augment	overall.	augment	overall.	augment	overall.	augment	overall
cage15	min.	133.85	146.52	7.42	34.47	27.29	40.75	0.22	14.07
	avg.	140.13	152.94	8.81	35.90	31.44	45.00	1.85	15.84
	max.	144.42	157.28	10.70	37.59	37.84	51.47	2.46	16.84
Channel-500	min.	64.29	74.46	9.15	29.81	12.18	22.62	0.04	6.65
	avg.	71.61	81.76	10.93	31.86	15.28	25.68	0.14	7.36
	max.	78.81	88.98	11.71	33.58	18.84	29.25	0.25	8.11
Circuit5M	min.	14.33	20.94	10.51	35.32	4.38	12.21	0.50	5.02
	avg.	15.26	22.01	13.11	38.04	5.70	13.09	0.77	5.80
	max.	16.00	22.72	14.42	39.43	6.81	13.68	1.31	7.79
Delaunay_24	min.	49.95	61.54	26.93	94.02	35.10	63.71	26.77	42.49
	avg.	54.79	66.40	29.99	98.29	36.68	64.70	31.06	46.81
	max.	61.23	72.81	32.70	104.13	40.30	68.11	34.09	49.77
Hugebubbles-0020	min.	68.17	83.14	55.79	148.64	44.83	82.31	42.02	62.56
	avg.	73.15	88.20	58.95	156.72	50.65	88.21	44.54	65.21
	max.	75.99	91.10	61.18	166.98	54.35	91.60	47.11	67.68

Table 2.8 – Detailed run times when MC21 is used for augmentations on the 39 real-life graphs from Section 2.3.3. The quality of heuristics are in Table 2.7. We have omitted graph `nlpkkt240` for which MC21 did not finish within a reasonable amount of time. For each instance we give the minimum, the average, and the maximum run time of five runs.

name	statistics	KS_{R1}		KS		2OUTMC		TRUNCRW	
		augment.	overall	augment.	overall	augment.	overall	augment.	overall
cage15	min.	2.19	14.89	2.11	29.18	1.90	15.46	0.73	14.22
	avg.	2.51	15.33	2.59	29.67	1.97	15.53	1.16	15.15
	max.	2.98	16.15	3.16	30.43	2.01	15.69	1.55	15.63
Channel-500	min.	1.70	11.84	1.82	22.50	1.19	11.60	0.04	6.66
	avg.	1.91	12.06	2.07	23.01	1.30	11.71	0.04	7.27
	max.	2.60	12.77	2.89	23.69	1.40	11.84	0.05	7.90
Circuit5M	min.	0.63	7.20	0.45	25.28	0.45	7.34	0.48	5.01
	avg.	0.77	7.53	0.62	25.55	0.53	7.93	0.58	5.61
	max.	0.97	7.97	0.90	25.92	0.67	9.55	0.64	7.04
Delaunay_24	min.	18.47	30.06	13.88	80.75	14.24	42.05	14.20	29.92
	avg.	20.83	32.44	14.89	83.19	15.47	43.49	17.67	33.41
	max.	22.33	33.91	16.17	86.35	17.12	44.98	20.40	36.09
Hugebubbles-0020	min.	23.09	38.09	14.99	106.41	23.27	60.75	21.97	42.54
	avg.	28.13	43.17	19.63	117.40	26.97	64.53	24.49	45.16
	max.	34.11	49.26	23.00	127.49	30.38	68.17	29.65	50.53
nlpkkt240	min.	27.01	125.69	28.19	210.27	14.91	97.26	13.76	77.87
	avg.	27.09	125.76	29.63	212.73	17.56	100.01	13.96	78.38
	max.	27.24	125.83	30.27	216.15	20.99	103.47	14.09	79.06

Table 2.9 – Detailed run times when PF+ is used for augmentations on the 39 real-life graphs from Section 2.3.3. The quality of heuristics are in Table 2.7. For each instance we give the minimum, the average, and the maximum run time of five runs.

name	statistics	KS_{R1} augment.	KS_{R1} overall	KS augment.	KS overall	2OUTMC augment	2OUTMC overall	TRUNCRW augment	TRUNCRW overall
cage15	min.	2.15	14.85	3.63	30.52	1.19	14.67	1.10	14.03
	avg.	2.41	15.22	3.80	30.88	1.39	14.95	1.28	15.28
	max.	2.68	15.85	4.01	31.08	1.69	15.32	1.69	16.59
Channel-500	min.	1.57	11.75	2.83	23.47	1.63	12.03	0.04	6.68
	avg.	1.66	11.81	2.92	23.86	1.75	12.16	0.06	7.28
	max.	1.70	11.85	3.01	24.88	2.02	12.44	0.08	7.92
Circuit5M	min.	116.67	123.24	107.51	132.34	2.02	8.89	0.74	5.26
	avg.	332.29	339.05	235.54	260.47	37.11	44.51	5.37	10.40
	max.	559.09	566.09	378.31	403.12	139.61	148.58	18.30	24.78
Delaunay_24	min.	40.52	52.15	32.09	98.89	41.66	69.52	48.63	64.32
	avg.	45.48	57.09	36.90	105.20	46.94	74.96	52.48	68.23
	max.	52.47	64.06	43.74	110.18	53.19	81.04	58.07	73.91
Hugebubbles-0020	min.	41.01	56.16	55.22	146.78	44.71	81.96	49.46	70.34
	avg.	47.53	62.58	58.56	156.33	51.59	89.15	53.16	73.84
	max.	52.59	67.56	61.17	166.57	58.54	96.15	54.82	75.36
nlpkkt240	min.	13.98	112.59	22.87	205.18	15.49	97.63	19.74	84.26
	avg.	14.13	112.80	24.17	207.27	17.34	99.79	28.70	93.13
	max.	14.51	113.27	25.77	211.10	19.01	101.46	47.31	112.28

Table 2.10 – Detailed run times when PR is used for augmentations on the 39 real-life graphs from Section 2.3.3. The quality of heuristics are in Table 2.7. For each instance we give the minimum, the average, and the maximum run time of five runs.

d	10000×10000 $\mathcal{M}^*$	10000×10000 TRUNCRW	12000×10000 $\mathcal{M}^*$	12000×10000 TRUNCRW
2	7787	0.9888	8724	0.9919
3	9266	0.9697	9667	0.9958
4	9761	0.9828	9899	0.9995
5	9918	0.9922	9973	1.0000

Table 2.11 – The quality of TRUNCRW on bipartite graphs without perfect matchings.

routines in exact matching algorithms.

2.3.3.4 With TRUNCRW

Here we perform a set of experiments focusing exclusively on TRUNCRW.

Experiments on matrices without total support. We experimented with bipartite graphs without total support which correspond to square (10000×10000) and rectangular matrices (12000×10000) with a uniform nonzero distribution. These matrices are generated with `sprand` command of Matlab and have about $d \times 10000$ nonzeros for $d = 2, 3, 4, 5$. The matrix representation of the bipartite graphs were scaled with 10 iterations of SK. For each d, we created five random matrices and ran TRUNCRW on the corresponding five instances. We report the worst quality of the five instances in Table 2.11. As seen in this table, TRUNCRW works just fine for this case. We did not report in the table but with increased SK iterations, the results improve, which is in accordance with earlier work [38].

Engineering TruncRW. Recall that TruncRW tries to find an augmenting path starting from a column vertex a certain number of times before giving up and moving to the next column vertex. We tested the behavior of TruncRW with different number of attempts at finding an augmenting path. Our test-set consisted of the 39 real-life instances from Section 2.3.3 and we tested with SK-5. When we allowed TruncRW just a single attempt, it was unable to find a perfect matching in any of the cases, and its average matching quality was 0.9984. When we allowed five attempts, TruncRW found a perfect matching for 13 graphs, and its average matching quality was 0.9999. With 10 attempts, it managed to find a perfect matching in 5 additional graphs. This verifies that allowing more attempts indeed improves the performance of the algorithm. The drawback, however, was the increased run time, which we did not think worth. That is why our implementation of TruncRW starts a random walk from a vertex only once.

We also test the effects of the look-ahead mechanism. Let us define the walk efficiency of TruncRW as the ratio of the cardinality of the matching found to the total length of the random walks. The higher this ratio, the more useful the random walks are. We evaluate the walk efficiency on a set of seven instances (real-life instances having at most 10000000 edges). We test both with and without scaling and report the results of the 14 tests. In 13 cases, the look-ahead mechanism improved the walk efficiency. The geometric mean (of 14 cases) of the ratios of walk efficiencies with look-ahead to that of without was 1.37. In the case where the look-ahead did not help (ratio was 0.71 in an instance named `Hamrle3`), the maximum deviation of a row or column sum from one after SK-5 was 0.28, which is high. We conclude that the look-ahead mechanism is very helpful.

Finally we test the effects that the length of the augmenting walk has on TruncRW. We doubled the allowed length of a random walk to $4(4 + 2n/(n - j))$. On average, the matching quality rose from 0.9984 to 0.9998. This modification was not able to find a perfect matching in any of the 39 instances. This led to an increase in the run time, which we deemed too large. We therefore keep $2(4 + 2n/(n - j))$ as the truncation length.

2.3.3.5 With 2outMC

Here, we briefly discuss the effects that the above two heuristics have on the performance of the 2outMC algorithm. Since 2outMC obtains high quality results, the two heuristics can only yield a relatively small improvement. When they are enabled and used with SK-5 2outMC finds matchings with average quality of 0.9997 for the real-life graphs from Section 2.3.3 for which 2outMC obtained matchings of quality 0.9983. This difference corresponds to about 13113 additionally matched edges, and hence signals that 13113 augmentations are avoided.

It is also interesting to consider the effects that these heuristics can have on cases where 2outMC did not deliver near-optimal matchings. As an example, we consider the synthetic family $\mathcal{J}'$ from Section 2.3.3. When scaling was not enabled, 2outMC found matchings of average cardinality $0.80 - 0.81\%$ of the maximum. If however one uses the two heuristics proposed in this section, then there is a significant improvement in performance, and 2outMC finds matchings of cardinality 0.89 of the maximum.

2.4 A scaling based derandomized algorithm

In the third part of this chapter, we discuss a derandomization of the OneSided heuristic. Our aim is to obtain a worst-case linear time, deterministic approximation algorithm with an

approximation guarantee strictly greater than $1/2$ and without augmenting path searches. The common point between this part and the previous one is the use of matrix scaling in order to produce reliable and efficient algorithms. The parts differ however in that while previously the scaled values were used to perform random decisions, here they will only be used to assist in deterministic decisions.

We first begin with some details about OneSided. Recall from Section 2.1 that OneSided is a linear time approximation algorithm for finding a matching in a bipartite graph with an approximation guarantee of $1 - 1/e$. We provide its details in Algorithm 2.3.

Algorithm 2.3: OneSided: Matching heuristic

Input: A bipartite graph $G = (V, E)$ and its matrix representation $\mathbf{A}$.
Output: A matching of G.

1: $[\mathbf{R}, \mathbf{C}] \leftarrow$ Scale($\mathbf{A}$)
2: **for** $i = 1$ to n **do**
3: Select a column j with probability s_{ij}, where s_{ij} is the corresponding entry from the scaled matrix $\mathbf{S} = \mathbf{RAC}$
4: match row i with column j

As in the heuristics discussed in the previous part of this chapter, OneSided first scales the adjacency matrix of the given graph in a preprocessing step in Line 1. The scaling procedure is only run for a few iterations such that the complexity of this step is linear for practical usage. Then it proceeds to select randomly for each row a neighbor column according to the values in that row. A column j can be selected by many rows in Line 3. In this case, we assume that j is matched with the row that selected it last. The use of scaling makes possible to attain the theoretical guarantee given in Theorem 2.5.

Theorem 2.5. *Algorithm 2.3 finds a matching of expected cardinality at least* $(1 - 1/e) \cdot n$.

We repeat the original proof for convenience and for its ties with the current work.

Proof. Let M be a random variable that represents the cardinality of the returned matching. Let random variable X_j be 1 if column j is matched or not. By the linearity of expectation, $\mathbb{E}[M] = \sum_{j=1}^{n} \mathbb{E}[X_j]$. The probability that X_j is equal to 1 is given by $\Pr(X_j = 1) = 1 - \prod_{i=1}^{n}(1 - s_{ij})$. That is, we consider the complement of the event that no row selects column j.

Now, we apply the arithmetic-geometry equality to derive the following upper bound on $\prod_{i=1}^{n}(1 - s_{ij})$ (and consequently a bound for $\Pr(X_j = 1)$)

$$\prod_{i=1}^{n}(1 - s_{ij}) \leq \left(\frac{\sum_{i=1}^{n}(1 - s_{ij})}{n}\right)^n \leq \left(\frac{n-1}{n}\right)^n \leq \left(1 - \frac{1}{n}\right)^n \approx 1/e \,. \tag{2.1}$$

We apply Equation (2.1) on the formula of expectation to conclude the proof

$$\mathbb{E}[M] = \sum_{j=1}^{n} \mathbb{E}[X_j] = \sum_{j=1}^{n} \Pr(X_j = 1) \geq (1 - 1/e) \cdot n \,.$$

□

2.4.1 The derandomization

We now present the derandomized algorithm for OneSided. Our result follows the technique of derandomizing based on conditional expectations. It bases its deterministic decisions in the idea of maximizing the expected outcome.

Let $\mathbb{E}[M|c_1, \ldots, c_{i-1}]$ represent the expected cardinality of a matching when the first $i-1$ rows have selected their columns. The choice c_z in particular denotes the column chosen by the zth row. When $i = 1$, we have to abuse slightly the notation such that $\mathbb{E}[M|c_1, \ldots, c_{i-1}] = \mathbb{E}[M]$. Our aim will be to select a column c_i for the ith row deterministically by considering the values of all possible $\mathbb{E}[M|c_1, \ldots, c_{i-1}, c_i]$.

Proposition 2.2. *For any $i \geq 1$, there exists a column j such that*

$$\mathbb{E}[M|c_1, \ldots, c_{i-1}] \leq \mathbb{E}[M|c_1, \ldots, c_{i-1}, j]$$

Proof. Let us assume the contrary. Then for any j, $\mathbb{E}[M|c_1, \ldots, c_{i-1}, j] < \mathbb{E}[M|c_1, \ldots, c_{i-1}]$. We now develop $\mathbb{E}[M|c_1, \ldots, c_{i-1}]$ as

$$\begin{aligned}\mathbb{E}[M|c_1, \ldots, c_{i-1}] &= \sum_{j=1}^{n} \mathbb{E}[M|c_1, \ldots, c_{i-1}, j] \cdot s_{i,j} \\ &< \mathbb{E}[M|c_1, \ldots, c_{i-1}] \cdot \sum_{j=1}^{n} s_{ij} \\ &< \mathbb{E}[M|c_1, \ldots, c_{i-1}] \ .\end{aligned}$$

We arrive thus at a contradiction. □

Our derandomized algorithm will hence pick at each step the column j that maximizes the value $\mathbb{E}[M|c_1, \ldots c_{i-1}, j]$. Proposition 2.2 allows us to show that the derandomized algorithm following this principle preserves the theoretical guarantee shown in Theorem 2.5 and hence will return a matching with expected cardinality at least $(1 - 1/e) \cdot n$.

It remains to see how to define and calculate the different $\mathbb{E}[M|c_1, \ldots c_i, j]$ values so as to make the decision c_i. Let M_{i-1} be the set containing all matched columns and U_{i-1} contain all the unmatched ones after $i-1$ steps of the algorithm. The expectation of the cardinality of the returned matching, based on M_{i-1} and the decision j of the ith row, can be written as

$$\mathbb{E}[M|c_1, \ldots, c_{i-1}, j] = \begin{cases} |M_{i-1}| + \sum_{z \in U_{i-1}} (1 - \prod_{q=i+1}^{n} (1 - s_{qz})) & j \in M_{i-1} \\ |M_{i-1}| + 1 + \sum_{z \in U_{i-1}, z \neq j} (1 - \prod_{q=i+1}^{n} (1 - s_{qz})) & j \in U_{i-1} \end{cases} . \tag{2.2}$$

If row i selects a matched column from the set M_{i-1}, then $M_i = M_{i-1}$ and likewise $U_i = U_{i-1}$. Otherwise, assuming row i selects column $j \in U_{i-1}$, we have $M_i = M_i \cup \{j\}$ and $U_i = U_{i-1} \setminus \{j\}$. We can now derive the formula for expected cardinality of the returned matching (2.2) by noting that the columns in M_i have been matched already hence their contribution to the expected cardinality of the returned matching is 1. The probability for a column $u \in U_i$ to be matched (i.e., $X_u = 1$) can be calculated in an equivalent way as in Theorem 2.5. We consider the remaining rows $i+1, \ldots, n$ and take the complement of the event that none of these rows will select u randomly.

A straightforward implementation of the above idea gives an $\mathcal{O}(mn^2)$ algorithm for a bipartite graph with n vertices per side, and m edges. For each nonzero of the matrix, we need to perform

$\mathcal{O}(n)$ calculations, each of which requires $\mathcal{O}(n)$ time. We now show how to reduce the complexity to linear time using Propositions 2.3 and 2.4. At first, we show that row i has to choose a neighbor from the U_{i-1} set to maximize the expected cardinality of the returned matching.

Proposition 2.3. *Let $j \in M_{i-1}, j' \in U_{i-1}$. Then $\mathbb{E}[M|c_1, \dots c_i, j] \leq \mathbb{E}[M|c_1, \dots c_i, j']$.*

Proof. It follows by noting that $1 \geq 1 - \prod_{q=i+1}^{n}(1 - s_{qj'})$. □

As a corollary, Proposition 2.3 shows that the matching returned by the derandomized algorithm is in fact maximal. Row i will select always an unmatched column if possible. This is in contrast to the original OneSided algorithm, which did not return necessarily maximal matchings. Proposition 2.4 discusses a method to compare between the vertices in the U_{i-1} set without having to calculate the expectation directly.

Proposition 2.4. *Let $j, j' \in U_{i-1}$ where column j leads to a higher expectation than j'. Then*

$$\prod_{q=i+1}^{n}(1 - s_{qj'}) \leq \prod_{q=i+1}^{n}(1 - s_{qj}) \,.$$

Proof.

$$\mathbb{E}[M|c_1, \dots c_i, j'] \leq \mathbb{E}[M|c_1, \dots, c_i, j]$$

$$|M_{i-1}| + 1 + \sum_{z \in U_{i-1}, z \neq j'} (1 - \prod_{q=i+1}^{n}(1 - s_{qz})) \leq |M_{i-1}| + 1 + \sum_{z \in U_{i-1}, z \neq j} (1 - \prod_{q=i+1}^{n}(1 - s_{qz})) \quad \text{by Eq. (2.2)}$$

$$\sum_{z \in U_{i-1}, z \neq j'} (1 - \prod_{q=i+1}^{n}(1 - s_{qz})) \leq \sum_{z \in U_{i-1}, z \neq j} (1 - \prod_{q=i+1}^{n}(1 - s_{qz}))$$

$$1 - \prod_{q=i+1}^{n}(1 - s_{qj}) \leq 1 - \prod_{q=i+1}^{n}(1 - s_{qj'}) \quad j, j' \text{ appear on different sides}$$

$$\prod_{q=i+1}^{n}(1 - s_{qj'}) \leq \prod_{q=i+1}^{n}(1 - s_{qj}) \,.$$

□

Proposition 2.4 allows us to see that whenever $\mathbb{E}[M|c_1, \dots, c_i, j] \geq \mathbb{E}[M|c_1, \dots, c_i, j']$ holds, then $\prod_{q=i+1}^{n}(1 - s_{qj}) \geq \prod_{q=i+1}^{n}(1 - s_{qj'})$ holds as well. This already reduces the complexity from $\mathcal{O}(m \cdot n^2)$ to $\mathcal{O}(m \cdot n)$, but it is still too high. To further bring down the complexity to $\mathcal{O}(m)$ in a preprocessing step we calculate all possible $\prod_{z=i+1}^{n}(1 - s_{z,j})$ values in a suffix-product fashion.

We summarize the OneSidedDerand algorithm in Algorithm 2.4. Similar to Algorithm 2.3, it begins by scaling the adjacency matrix of the given bipartite graph for a few steps. It then calculates in $\mathtt{P}[:, j]$ for each column j the suffix product array of $\prod_{i=1}^{n}(1 - s_{ij})$ using an intermediate array $\mathtt{L}$. Then, starting with the first row, each row picks the unmatched neighboring column that maximizes the corresponding value in the $\mathtt{P}$ array. If a row has no free columns, it is skipped. For further efficiency, we can remove the $\mathtt{P}$ array and base the decision at Line 13 exclusively on the $\mathtt{L}$ array. To do so, once we are finished with the selection of the ith row, we update $\mathtt{L}[j] = \dfrac{\mathtt{L}[j]}{1 - s_{ij}}$ for each nonzero neighbor j of the ith row. One can then easily verify

Algorithm 2.4: OneSidedDerand: The derandomized variant of OneSided

Input: A bipartite graph $G = (V, E)$ and its matrix representation $\mathbf{A}$.
Output: A matching of G.

```
[R, C] ← Scale(A)
for i = 1 to n do
    L = [1, ..., 1]
    for i = n down to 1 do
        for j ∈ adj(i) do
            P[i, j] ← L[j]
            L[j] ← L[j] · (1 − s_ij) , where s_ij is the corresponding entry from
            the scaled matrix S = RAC
U_0 ← {1, ..., n}
M_0 ← ∅
for i = 1 to n do
    U ← adj(i) ∩ U_{i−1}
    if U ≠ ∅ then
        c_i ← arg max_{j∈U} P[i, j]
        match row i with column j
        U_i ← U_{i−1} \ c_i
        M_i ← M_{I−1} ∪ {c_i}
    else
        U_i ← U_{i−1}
        M_i ← M_{i−1}
```

that at the beginning of the ith step the values of the L array will be identical to $\mathtt{P}[i,:]$, i.e., $\mathtt{L}[j] = \mathtt{P}[i,j]$ for all unselected columns j.

While Algorithm 2.4 exhibits good behavior (see Section 2.4.2), improvements on its performance are still possible. At Line 13 of the algorithm, the ith row picks the column which is the least likely to be selected by the rest of the remaining rows. The sampling probabilities for each row were defined only once, during Line 1 and do not accurately represent the current state of the graph. As we discussed, the ith row will avoid vertices from M_{i-1}. It would instead be more appropriate to use nonzero probabilities only for the values corresponding in unmatched columns in (2.2). In this case, each row would zero-out any matched columns and normalize the rest of the entries such that the row continues to define a probability distribution. Unfortunately, updating the probabilities every time a new vertex is matched yields an $\mathcal{O}(mn)$ algorithm which can highly inefficient. We are not aware of a more efficient method to update the values of the matrix in the described way.

We opted to handle degree-1 vertices in a way similar to Rule-1 in Karp–Sipser. In this approach, we remove vertices from the graph along with their edges when they are matched. Rows select columns as in Algorithm 2.4 by comparing between expectations implicitly, except when degree-1 vertices appear. In this case, we include the degree-1 vertex and its neighbor to the matching. We call the resulting algorithm OneSidedDerand$_{R1}$. As we discussed in Section 2.2, these decisions do not harm the matching cardinality. Note that this variant does not work in conjunction with the P array. The P array assumes that the rows always choose in the order $1, \ldots, n$, which might not be necessarily true in this case. The version relying solely on the L array should be preferred instead. Rule-2 of Karp–Sipser can be adapted in a similar manner.

2.4.2 Some preliminary experiments

We give the results of some experiments showcasing the promising performance of the OneSidedDerand heuristic. We selected 171 medium-sized matrices with $10^4 \leq n \leq 3 \cdot 10^5$ from the SuiteSparse Matrix Collection [31]. We chose to compare OneSidedDerand against the KS$_{R1}$ heuristic that only applies Rule-1 of Karp–Sipser as both of these algorithms run in linear time. Other than OneSidedDerand, we additionally considered its variation OneSidedDerand$_{R1}$ that applies Rule-1 of Karp–Sipser whenever it is possible. The results can be seen in Figure 2.11, where we show the approximation ratio of the cardinality returned by the different algorithms versus the maximum cardinality.

As can be seen, OneSidedDerand$_{R1}$ has overall the best performance. While in most cases KS$_{R1}$ fares better than OneSidedDerand, we note that on average the quality of the latter is in fact higher. OneSidedDerand never drops below in approximation 0.94 whereas the worst approximation of KS$_{R1}$ is about 0.85, which is significantly worse.

The original OneSided heuristic is known to obtain results around its theoretical bound [38]. In contrast, the performance of OneSidedDerand and its variant is nearly 30% better than its theoretical guarantee. OneSided in particular achieves the $(1 - 1/e)$-approximation bound on complete bipartite graphs, where OneSidedDerand however is optimal. Due to the these, we suspect that the actual theoretical guarantee of the derandomized heuristics can potentially be higher. Note that one can similarly make the matching of OneSided maximal in a follow-up step thereby improving its approximation. Including Rule-1 consistently improves the behavior of OneSidedDerand. The difference can be as high as 4%. This can be explained by the discussion in the previous section about how Algorithm 2.4 works with outdated probabilities.

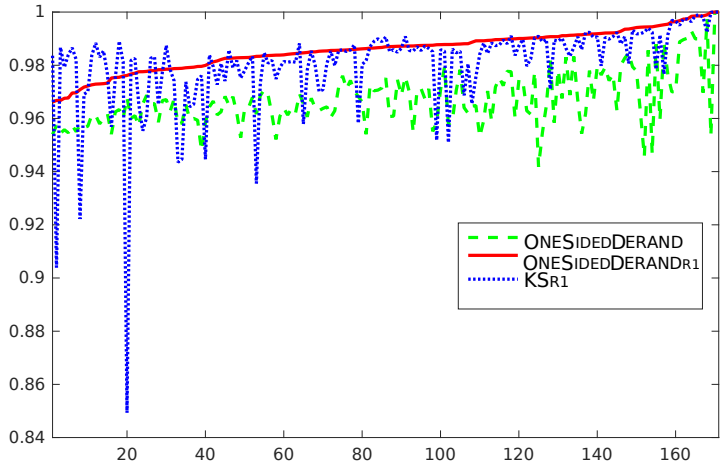


Figure 2.11 – Comparisons between the different versions of OneSidedDerand. The y axis corresponds to the approximation ratios of the three algorithms. The results are sorted with respect to the approximation ratio of the derandomized version using Rule-1.

2.5 Concluding remarks

This chapter focused on heuristics for bipartite graph matching and comprised of three parts. While in all parts we dealt with efficient heuristics, our motivation in each part was different. The first part dealt with the efficient implementation of the Karp–Sipser heuristic. We were motivated in here to examine the second rule of Karp–Sipser as it has not been as extensively studied as the first rule. We investigated it both complexity-wise as well as from an experimental point of view to study its effectiveness in practice. The second part considered randomized heuristics based on matrix scaling. As we discussed, heuristics such as Karp–Sipser have been known to behave poorly for some instances. Our motivation hence in this part was to produce fast, robust, and reliable heuristics that could overcome such limitations. We achieved this goal by utilizing matrix scaling to assist with the random decisions. In the third part, we proposed a deterministic approximation algorithm again based on matrix scaling. We were motivated by the fact that the deterministic approaches known to surpass the 1/2 bound in approximation are based on augmenting paths. We therefore became interested to see whether our matrix scaling framework would translate well into an effective non probabilistic algorithm that can obtain higher than 1/2 approximation without resorting to augmenting path searches.

For our results in the first part, we investigated two data reduction rules for the maximum cardinality matching problem proposed by Karp and Sipser [72]. While the first rule has a simple, worst case linear time implementation, the second rule can take quadratic time. We focused on the second rule which merges the two neighbors of a degree-2 vertex. We considered and analyzed three different algorithms with different levels of sophistication, and proposed an efficient algorithm to recover the matching in the original graph. For two of the these algorithms, we showed that their worst-case performance can still be quadratic in terms of n, whereas the third approach has sub-quadratic complexity in sparse graphs. On a set of experiments with real-life, random, and constructed problem instances we showed that the second rule indeed increases the cardinality of the matching found by Karp–Sipser and can lead to a drop in the run time of the exact algorithm afterwards. One open question is whether a linear time implementation for the second rule is actually possible. As discussed in the text, it seems that the approach of

Bartha and Krezs [7] does not achieve a linear time complexity when degree-2 reductions are interleaved with degree-1 reductions or random choices, made in the matching context. With these, we conjecture that the answer to our question is negative. We are also interested in a parallel version of the Karp–Sipser algorithm. To that end, we believe that the component-based approach discussed in Section 2.2.3 should adapt very well to parallel settings.

In the second part of the chapter, we examined two randomized algorithms for the maximum cardinality matching problem in bipartite graphs. These algorithms work with two special classes of bipartite graphs and we discussed how to generalize them through the use of matrix scaling. Our first algorithm 2OUTMC is based on a Monte-Carlo algorithm for matching in 2-out random subgraphs [71]. The second algorithm TRUNCRW is based on a Las Vegas algorithm for matching in d-regular bipartite graphs [54]. Our experimental results showed that these approaches obtain near perfect matchings in real-life and synthetic instances and have a near linear time run time. The two approaches were also shown to be more robust than the state of the art heuristics used in the cardinality matching algorithms, and are generally more useful as initialization routines. Our adaptation of 2OUTMC is based on the premise that 2-out graphs sampled from a host graph have perfect matchings, assuming that the matrix representation of the host graph have total support. We showed empirical evidence that k-out graphs, for $k \geq 3$, sampled from a host graph with total support always contain perfect matchings. For $k = 2$, we noticed the existence of perfect matchings in many graphs. A proof or the disproof of such k-out graphs having perfect matchings is certainly welcome. Furthermore, this was the first attempt to implement 2OUTMC, and there is room for improved performance.

In the third part, we discussed a deterministic matrix scaling based heuristic ONESIDED-DERAND and its variant ONESIDEDDERAND$_{R1}$ which derandomize the ONESIDED heuristic [38]. These two deterministic algorithms have the same complexity and theoretical guarantees as ONESIDED. On a large set of medium-sized bipartite graphs, we noticed that their quality greatly surpasses their theoretical guarantee. For this reason, we would like to analyze the behavior of the ONESIDEDDERAND algorithm in more detail. Furthermore, we would like to derandomize other randomized heuristics for matching such as the ones discussed in Section 2.1.

Chapter 3

Matchings in undirected graphs

In this chapter we investigate the problem of matching in undirected graphs. This chapter can thus be thought of as a continuation of the previous chapter. We again examine algorithms that can find large matchings reasonably fast for the purpose of speeding up exact matching algorithms or for use in applications which require large matchings [89]. The results of this chapter were published in the proceedings of the CSC 18 workshop [W1].

Our main contribution in the chapter is adapting the TwoSided herustic [38] to general undirected graphs. Recall from Section 2.1, that the TwoSided heuristic is a fast practically linear time algorithm that achieves a 0.866-approximation on bipartite graphs with total support. This approximation ratio is achieved through the use of matrix scaling by generating an 1-out random subgraph of the given bipartite graph. While the proposed heuristic is a straightforward adaptation of its counterpart for bipartite graphs, its analysis is more complicated due to the existence of odd cycles in the generated subgraph. Our analysis shows that a random 1-out subgraph of a given graph has maximum cardinality of a matching around $0.866 - \log(n)/n$ of the best possible—the observed performance is higher. We also discuss two variants of the main heuristic without proofs. Both of these variants enrich a 1-out subgraph with more edges, and hence deliver better results than the main heuristic both in theory and in practice.

The rest of the chapter is organized as follows. In Section 3.1 we present the main heuristic, its analysis as well as its two variants. In Section 3.2 we provide the experimental results. Section 3.3 summarizes the chapter and discusses some potential future work.

3.1 One-Out: The main heuristic

The One-Out heuristic shown in Algorithm 3.1 first scales the adjacency matrix of a given undirected graph to be doubly stochastic. The heuristic then selects a random vertex randomly for each of the vertices based on the values in the scaled matrix. Then, the subgraph of the original graph containing only the selected edges is formed, yielding an undirected graph with at most n edges, each vertex having at least one edge incident on it. The KS_{R1} heuristic discussed in the previous chapter that only applies Rule-1 of Karp–Sipser and random decisions is then run on this subgraph. Due to the fact that the 1-out graph consists only of unicyclic components, KS_{R1} is capable of finding a maximum matching using only degree-1 reductions and random selections [38, Lemma 3]. The approximation guarantee of the proposed heuristic is analyzed until this step.

The original TwoSided heuristic performed only the equivalent of Lines 1 until 6 from Algorithm 3.1. At Line 7, we have included an additional step to improve its performance by

ensuring that the returned matching is maximal. Notice that the maximum matching in the 1-out subgraph is not necessarily maximal for the entire graph. To achieve maximality, we run KS_{R1} or another GREEDY-like heuristic on the subgraph of all unmatched vertices, which naturally contains only edges that were not selected in the previous steps. As we will see, this addition brings in a large improvement to the cardinality of the matching in practice.

Algorithm 3.1: ONE-OUT: Heuristic for matching in undirected graphs

Input: $G = (V, E)$ and its adjacency matrix $\mathbf{A}$.
Output: `match`$[\cdot]$: the matching.

1: $\mathbf{R} \leftarrow$ SYMSCALE($\mathbf{A}$)
2: **for** $i = 1$ **to** n **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 $s_{ik} = \mathbf{R}[i] \times \mathbf{R}[k]$ is the corresponding entry in the scaled matrix $\mathbf{S} = \mathbf{RAR}$.
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

Let us consider that the edges incident on a vertex v_i consist of 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).

The analysis traces an execution of KS_{R1} on the subgraph G^1_{out} and we identify two possible phases. KS_{R1} starts with *Phase-1* which continues for as long as there exist degree-1 vertices in the graph. *Phase-2* thus starts at the first moment that KS_{R1} has to rely on a random decision. We now introduce some notation which will prove helpful in the analysis of the algorithm during Phase-1. Let A_1 be the set of vertices not chosen by any other vertex at Line 4 of Algorithm 3.1. These vertices have in-degree zero and out-degree one, and hence can be processed by KS_{R1}. 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, for $i \geq 1$. Notice that $A_i \subseteq A_{i+1}$ and $B_i \subseteq B_{i+1}$. The KS_{R1} 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 KS_{R1} instrumented to highlight them.

Lemma 3.1. *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 3.1. *$A_i \cap B_j = \emptyset$ 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 3.1. □

Thanks to Lemma 3.1 and Corollary 3.1, we have that 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 version of the KS_{R1} heuristic for 1-out graphs is shown in Algorithm 3.2. In practice, Algorithm 3.2 is a straightforward adaptation of KS_{R1}, extending it only by keeping some information related to the analysis of ONE-OUT to ease understanding. For simplicity, we will use the term KS_{R1} to refer to Algorithm 3.2 in the text below. The degree-1 vertices are kept in a first-in first-out priority queue Q. The queue is first initialized with A_1, and a # 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 1 to the queue. During Phase-1 of KS_{R1}, 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 3.1 has linear worst-case time complexity of $\mathcal{O}(n + m)$ and linear space requirements, using the CSC format. In regards to the cost of the scaling step in Line 1, we follow the same methodology as in our heuristics for bipartite graph matching in the previous chapter. We hence run the scaling algorithm only for a constant number of steps. The only requirement is a scaling method which can preserve symmetry. In practice, 5 or 10 iterations of the basic method [80] or even less of the Newton iterations [79] seem sufficient (see experiments). Therefore, the practical run time of the algorithm is linear.

3.1.1 ONE-OUT: Analysis

Let a_i and b_i be two random variables representing the cardinalities of A_i and B_i, respectively, in an execution of KS_{R1} on a random 1-out graph. Then, the KS_{R1} algorithm matches b_ℓ edges in Phase-1, and leaves $a_\ell - b_\ell$ vertices unmatched. What remains after Phase-1 is a set of cycles. In the bipartite graph case [38], each of these cycles has even length. Therefore all vertices in those cycles are matchable and hence the cardinality of the matching is measured by $n - a_\ell + b_\ell$. Since we can possibly have odd cycles after Phase-1, 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 Phase-1 of KS_{R1}. Then we have the following lemma which we give without proof about the approximation guarantee of Algorithm 3.1 after the completion of Line 6.

Lemma 3.2. *At the end of execution, the number of unmatched vertices is $a_\ell - b_\ell + c$. Hence, Algorithm 3.1 matches at least $n - (a_\ell - b_\ell + c)$ vertices.*

We need to quantify $a_\ell - b_\ell$ and c in Lemma 3.2. Initially, we obtain an upper bound on $a_\ell - b_\ell$. Afterwards, we at first obtain a pessimistic upper bound on c, which we then improve with a

Algorithm 3.2: KARPSIPSER$_{\text{ONE-OUT}}$: Specialized Karp–Sipser for 1-out graphs

Input: $G^1_{out} = (V, E)$: a 1-out graph.
Output: `match`$[\cdot]$: the mates of vertices.
Output: ℓ: the number of levels in Phase-1.
Output: A: the set of degree-1 vertices in Phase-1.
Output: B: the set of vertices matched to A vertices.

```
match[u] ← NIL for all u
Q ← {v : deg(v) = 1}{degree-1 or in-degree 0}
if Q = ∅ then
    ℓ ← 0{no vertex in level 1}
else
    ℓ ← 1
ENQUEUE(Q, #){marks the end of the first level}
Phase-1← ongoing
A ← B ← ∅
while true do
    while Q ≠ ∅ do
        u ← DEQUEUE(Q)
        if u = # and Q = ∅ then
            break the while-Q-loop
        else if u = # then
            ℓ ← ℓ + 1
            ENQUEUE(Q, #){new level formed}
            skip to the next while-Q-iteration
        if match[u] ≠ NIL then
            skip to the next while-Q-iteration
        for v ∈ adj(u) do
            if match[v] = NIL then
                match[u] ← v
                match[v] ← u
                if Phase-1 = ongoing then
                    A ← A ∪ {u}
                    B ← B ∪ {v}
                N ← adj(v)
                G^1_out ← G^1_out \ {u, v}
                ENQUEUE(Q, w) for w ∈ N, deg(w) = 1
                break the for-v-loop
        if Phase-1 = ongoing and match[u] = NIL then
            A ← A ∪ {u}{u cannot be matched}
    Phase-1 ← done
    if E ≠ ∅ then
        pick a random edge (u, v)
        match[u] ← v
        match[v] ← u
        N ← adj({u, v})
        G^1_out ← G^1_out \ {u, v}
        ENQUEUE(Q, w) for w ∈ N, deg(w) = 1
    else
        break the while-true loop
```

more detailed analysis. While the bound for $a_\ell - b_\ell$ holds for any graph with total support presented to Algorithm 3.1, 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 3.1.

Theorem 3.1. *Algorithm 3.1 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.*

The theorem implies that 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 [38]. 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 3.2.1, we empirically show that the same bound also holds for graphs whose corresponding matrices do not have total support.

3.1.1.1 An upper bound on $a_\ell - b_\ell$

In order to measure $a_\ell - b_\ell$, we adapt a proof from earlier work [38], which was inspired by Karp and Sipser's analysis of the first phase of their heuristic [72]. Let $\alpha_j^{(k)} = \Pr(v_j \in A_k)$ be the probability for a vertex v_j to belong to A_k, and similarly $\beta_j^{(k)} = \Pr(v_j \in B_k)$ be the probability for a vertex v_j to belong to B_k.

Lemma 3.3. *It holds that*

$$\beta_j^{(k)} \geq \quad 1 - e^{-\sum_i s_{ij} a_i^{(k)}} \tag{3.1}$$

$$\alpha_j^{(k)} \leq \quad e^{1-\sum_i s_{ij} \beta_i^{(k-1)}} \ . \tag{3.2}$$

The proof of the lemma can be derived by following the bipartite case [38, Lemmas 6 and 7]. That is why, we give a high level sketch.

Proof. (Sketch) By Lemma 3.1 and its corollary, A_ℓ and B_ℓ define a bipartition. Therefore, the equations remain the same. The only technicality in the proof is to make sure that the vertices in A_k and those in B_k are from the same set V and to take the bipartition into account. □

Thanks to Lemma 3.3, we can now bound $a_\ell - b_\ell$.

Lemma 3.4. $a_\ell - b_\ell \leq (2\Omega - 1)n$, *where* $\Omega \approx 0.567$ *equals to* $W(1)$ *of Lambert's* W *function.*

Proof. In expectation, $a_\ell = \sum_{i=1}^n \alpha_i^{(\ell)}$ and $b_\ell = \sum_{i=1}^n \beta_i^{(\ell)}$. The expected difference is

$$a_\ell - b_\ell = \sum_{i=1}^n \alpha_i^{(\ell)} - \beta_i^{(\ell)} \ .$$

From Lemma 3.3 and another result of Dufossé et al. [38, Lemma 8], we have $\alpha_i^{(\ell)} - \beta_i^{(\ell)} \leq (2\Omega - 1)$ and hence $a_\ell - b_\ell \leq \sum_{i=1}^n (2 \cdot 0.567 - 1) \ \leq 0.134 \cdot n$. □

3.1.1.2 An upper bound on c

We now investigate c, the number of odd cycles that remain on a G^1_{out} graph after Phase-1 of KS_{R1}.

Lemma 3.5. *We have* $c \leq \frac{n-a_\ell-b_\ell}{3}$.

Proof. After Phase-1, we have removed A_ℓ and B_ℓ from V. Therefore, the number of vertices remaining for the second phase is $n - a_\ell - b_\ell$. The shortest odd cycle is of length 3 and so we can derive the lemma by assuming that all vertices belong to such cycles. □

We need a lower bound on $a_\ell + b_\ell$ to bound c in Lemma 3.5. We start by bounding a_ℓ first.

Lemma 3.6. *For random 1-out graphs with* $n \geq 30$ *vertices,* $a_\ell \geq 0.361n$.

Proof. As $A_i \subseteq A_{i+1}$, we have $\alpha_i^{(\ell)} \geq \alpha_i^{(1)}$, hence $a_\ell \geq a_1$. The probability of a vertex u_i to be in A_1 is $\alpha_i^{(1)} = \left(1 - \frac{1}{n-1}\right)^{n-1}$, since u_i should not be chosen by any other vertex. For $n \geq 30$, we have $\frac{1}{e} \geq \left(1 - \frac{1}{n-1}\right)^{n-1} \geq 0.361$. This concludes the proof by the linearity of expectation. □

We can use the result of the above lemma to bound b_ℓ.

Lemma 3.7. *For random 1-out graphs with* $n \geq 30$ *vertices,* $b_\ell \geq 0.303n$.

Proof. We start again with $b_\ell \geq b_1$, as $B_i \subseteq B_{i+1}$. By Equation (3.1),

$$b_\ell \geq \sum_j 1 - e^{-\sum_i s_{ij}\alpha_i^{(1)}} ,$$

and by using Lemma 3.6,

$$b_\ell \geq \sum_j 1 - e^{-0.361} ,$$

for $n \geq 30$. This simplifies to the stated result. □

The proofs used in obtaining the bounds in Corollary 3.2 needed to assume a random 1-out graph only for lower bounding $\alpha_i^{(1)}$. We sum the two lower bounds from Lemmas 3.6 and 3.7 in a corollary.

Corollary 3.2. $a_\ell + b_\ell \geq 0.664n$ *for random 1-out graphs with* $n \geq 30$ *vertices.*

At this point, we have a bound on the number of matched vertices using Lemmas 3.2, 3.4, 3.5, and Corollary 3.2 as $n - a_\ell + b_\ell - \frac{n-a_\ell-b_\ell}{3} \geq 0.754n$.

3.1.1.3 An improved bound on c

We now discuss how a different analysis and show a tighter bound for c. The notion of derangements will prove useful in the analysis, and we remind the definition in here for convenience. A permutation of $1, \ldots, n$ in which no element stays in its original position is called *derangement*. The total number of derangements of $1, \ldots, n$ is denoted by $!n$.

We now consider the family of graphs in which all vertices have degree two. These graphs consist of disjoint cycles, and we will refer to them as *cyclic graphs*. Let C_M denote a random

graph from this family with M vertices. There are $!M$ cyclic graphs with M vertices, as derangements create cycles. The $n - a_\ell - b_\ell$ vertices remaining after Phase-1 of KS_{R1} form a $C_{n-a_\ell-b_\ell}$ graph by the principle of deferred decisions [92, p. 9]. This is so, as the edge chosen by a vertex u is incident on a remaining vertex, and u is chosen by exactly one of the remaining vertices. We will find an upper bound on the expected number of odd cycles in a random $C_{n-a_\ell-b_\ell}$ graph and improve Lemma 3.5.

For a vertex $u_i \in C_M$, let h_i be the length of the cycle containing it, and define a variable $Y_i = \frac{1}{h_i}$ if h_i is odd and $Y_i = 0$ otherwise. Then, $\sum_{u_i \in C_M} Y_i$ is the number of odd cycles in C_M. To see why, consider an odd cycle of length k and observe that each of its vertices contributes with $\frac{1}{k}$ for counting the cycle. We now measure the expected value of Y_i for a vertex u_i.

Lemma 3.8. *In a random cyclic graph with M vertices, $\Pr(h_i = k) \leq \frac{1.04}{M}$ for any vertex u_i when $k < M - 2$, and $\Pr(h_i = k) \leq \frac{1.365}{M}$ for $k = M - 2$.*

Proof. We have the formula whose explanation follows:

$$\Pr(h_i = k) = \frac{\binom{M-1}{k-1} \cdot (k-1)! \cdot !(M-k)}{!M} .$$

We select a set of $k - 1$ vertices which will form a cycle of length k with u_i. There are $\binom{M-1}{k-1}$ different sets, and there are $(k-1)!$ possible cycles with the selected set. The remaining $M - k$ vertices must also lie in cycles; there are $!(M-k)$ ways to achieve this. Furthermore, with M vertices there are $!M$ ways to create a union of disjoint cycles. Let us manipulate the formula as

$$\begin{aligned}\Pr(h_i = k) &= \frac{\binom{M-1}{k-1} \cdot (k-1)! \cdot !(M-k)}{!M} \\ &= \frac{(M-1)! \cdot !(M-k)}{!M \cdot (M-k)!} \\ &= \frac{1}{M} \cdot \frac{M! \cdot !(M-k)}{!M \cdot (M-k)!} .\end{aligned}$$

For $M \geq 5$, we have $\dfrac{M!}{!M} \leq 2.73$. We now distinguish between three cases depending on the value of $M - k$. If $M - k \geq 4$, $\dfrac{(M-k)!}{!(M-k)} \geq 2.64$, and we see that $\dfrac{M!}{!M}\dfrac{(M-k)!}{!(M-k)} \leq \frac{2.73}{2.64} \leq 1.04$. When $M - k = 3$, we have $\dfrac{3!}{!3} = 3$, and $\dfrac{M!}{!M}\dfrac{(M-k)!}{!(M-k)} \leq \frac{2.73}{3} \leq 1 \leq 1.04$. When $M - k = 2$, we have $\dfrac{2!}{!2} = 2$, and the upper bound is $2.73/2 = 1.365$. □

Lemma 3.9. *In a random cyclic graph with M vertices, $\mathbb{E}[Y_i] \leq \dfrac{1.04}{M} \cdot \log(M) + \dfrac{0.1625}{M}$ for a vertex u_i.*

Proof.

$$\begin{aligned}
\mathbb{E}[Y_i] &= \sum_{k=3,odd}^{M} \Pr(h_i = k) \cdot \frac{1}{k} \\
&\leq \sum_{k=1}^{M} \Pr(h_i = k) \cdot \frac{1}{k} \\
&\text{(by splitting } 1.365/M \text{ for } k = M-2) \\
&\leq \left(\sum_{k=1}^{M} \frac{1.04}{M} \cdot \frac{1}{k} \right) + \frac{(1.365 - 1.04)}{M} \cdot \frac{1}{M-2} \\
&\leq \frac{1.04}{M} \cdot \sum_{k=1}^{M} \frac{1}{k} + \frac{0.1625}{M} \\
&\leq \frac{1.04}{M} \cdot \log(M) + \frac{0.1625}{M} \ .
\end{aligned}$$

□

Lemma 3.10. *The number of odd cycles in C_M is less than or equal to $\lceil 1.04 \log(M) \rceil$.*

Proof. By the linearity of expectation, we show the result by summing $\mathbb{E}[Y_i]$ over all u_i in C_M. □

We plug in the number of vertices remaining after Phase-1 of KS_{R1} into the formula and obtain the following corollary.

Corollary 3.3. *The expected number of cycles remaining after the Phase-1 of KS_{R1} 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 $\lceil 1.04 \log(0.336n) \rceil$.*

3.1.2 Two variants of One-Out

Here we summarize two related theoretical random bipartite graph models that we adapt to the undirected case using similar algorithms. The presentation will be brief and without proofs; we will present experiments in Section 3.2.1.

The random $(1 + e^{-1})$-out undirected graph model extends the 1-out model in the following way. Initially it lets each vertex choose a random neighbor. Then, vertices which were not selected by other vertices are allowed to 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. This model is based on the original idea of M. Karoński and Pittel [69] which we discused in Section 1.1. Recall that the initial claim about such subgraphs having perfect matchings was recently disproved [68].

A model richer in edges is the random 2-out graph model [46], whose analogous for bipartite graphs [71, 107] we examined in detail during the previous chapter. In this model, each vertex chooses two of its neighbors. Contrary to the bipartite graph case, a specialized algorithm such as 2outMC for finding a perfect matching in random 2-out undirected graphs is not known even for the case where the host graph is complete. Note that 2outMC makes excessive use of the bipartiteness of a graph as it alternates through the vertex disjoint RG and CG multigraphs. Extending 2outMC for undirected graphs is thus not straightforward.

3.2 Experiments

To understand the effectiveness and efficiency of the proposed heuristics in practice, we report the matching quality and various statistics that appeared in our analysis of the algorithms in Section 3.1. Our focus in this chapter was on fast linear algorithms, hence we compared One-Out only against KS_{R1}.

For the experiments, we used three graph datasets: (1) the first set is generated with matrices from SuiteSparse Matrix Collection [31]. 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 ended with empty rows and columns. There were 115 matrices at the end which we used as the adjacency matrices of the graphs. (2) The graphs in the second dataset belong to $\mathcal{J}$, the family of synthetic graphs discussed in Section 2.3.3 of the previous chapter. For this test, we have five graphs from $\mathcal{J}$ with different hardness levels for KS_{R1}. (3) The third set contains five large, real-life matrices from the SuiteSparse collection for measuring the run time efficiency of the proposed heuristics.

3.2.1 On real-life graphs

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 3.1 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 KS_{R1} 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).

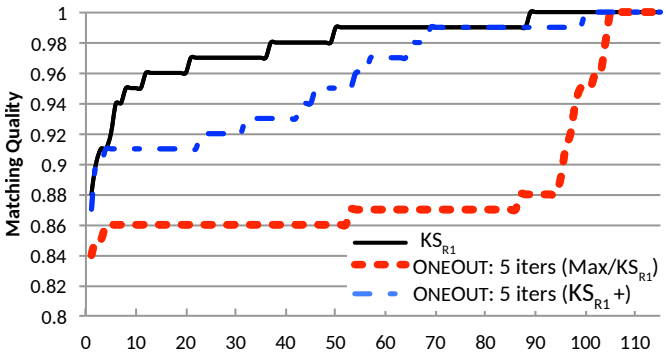
For the U-Max rows, we construct k-out graphs by using uniform probabilities while selecting the neighbors as proposed in the literature [46, 69]. We compute the cardinality of the maximum matchings in these k-out graphs to the maximum matching cardinality on the original graphs and report the statistics. 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_{R1} heuristics which execute KS_{R1} on the k-out graphs obtained with t scaling iterations. As we discussed, the matchings obtained by St-KS_{R1} heuristics are not maximal with respect to the original edgeset of the graph. The proposed St-KS_{R1}+ heuristics exploit this fact and apply another KS_{R1} 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 KS_{R1} is an optimal algorithm for these subgraphs.

As Table 3.1 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_{R1} vs U-Max) is significant. On the other hand, the difference on the matching quality for S5-KS_{R1} and S10-KS_{R1} is minor. Hence, five scaling iterations are deemed sufficient for the proposed heuristics in practice.

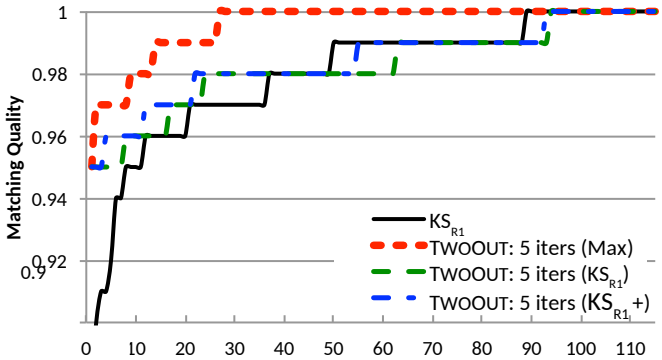
The heuristics St-KS_{R1} perform well for $(1 + e^{-1})$-out and 2-out graphs. With $t \in \{5, 10\}$,

	Alg.	Min	Max	Avg.	GMean	Med.	StDev
	KS_{R1}	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_{R1}	0.479	1.000	0.869	0.866	0.863	0.059
	S5-KS_{R1}	0.839	1.000	0.885	0.884	0.865	0.044
	S10-KS_{R1}	0.858	1.000	0.889	0.888	0.866	0.045
	S1-KS_{R1}+	0.836	1.000	0.951	0.950	0.953	0.043
	S5-KS_{R1}+	0.865	1.000	0.958	0.957	0.968	0.036
	S10-KS_{R1}+	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_{R1}	0.642	1.000	0.963	0.962	0.972	0.041
	S5-KS_{R1}	0.918	1.000	0.972	0.972	0.976	0.020
	S10-KS_{R1}	0.934	1.000	0.975	0.975	0.977	0.018
	S1-KS_{R1}+	0.857	1.000	0.972	0.972	0.979	0.025
	S5-KS_{R1}+	0.925	1.000	0.978	0.978	0.984	0.018
	S10-KS_{R1}+	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_{R1}	0.651	1.000	0.974	0.974	0.981	0.035
	S5-KS_{R1}	0.945	1.000	0.982	0.982	0.984	0.013
	S10-KS_{R1}	0.947	1.000	0.984	0.983	0.984	0.012
	S1-KS_{R1}+	0.860	1.000	0.980	0.979	0.984	0.020
	S5-KS_{R1}+	0.950	1.000	0.984	0.984	0.987	0.012
	S10-KS_{R1}+	0.952	1.000	0.986	0.986	0.987	0.011

Table 3.1 – For each matrix in the first dataset and each proposed heuristic, five runs are performed the statistics are performed 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 3.1 – The matching qualities of KS$_{R1}$ on the original graph, S5-KS$_{R1}$ and S5-KS$_{R1}$+ on 1-out and 2-out graphs for all 115 real-life graphs of Section 3.2.1. We have sorted these instances in increasing order of the quality of KS$_{R1}$. The figures also contain the quality of the maximum cardinality matchings in the generated 1-out and 2-out subgraphs.

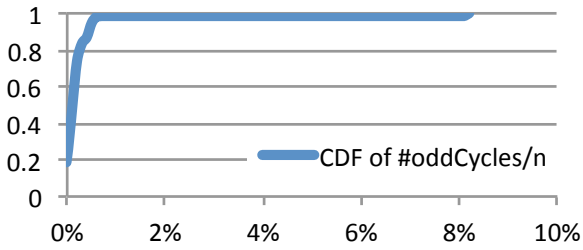


Figure 3.2 – The cumulative distribution of the ratio of number of odd cycles remaining after Phase-1 of KS_{R1} in One-Out to the number of vertices in the graph for the real-life graphs of Section 3.2.1.

their quality is almost on par with KS_{R1} on the original graph, and even better for 2-out graphs. In addition, applying KS_{R1} 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 is not significant. On the other hand, this extra step significantly improves the matching quality for 1-out graphs as on average they obtain the lowest matching quality.

To better understand the practical performance of the proposed heuristics and the impact of the additional KS_{R1} execution, we profile their performance by sorting their matching qualities in increasing order for all 115 matrices. Figure 3.1(a) 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 for 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 KS_{R1} can obtain on the original graph. The additional KS_{R1} in S5-KS_{R1}+ closes almost all of this gap and makes the matching qualities close to those of KS_{R1}. On the contrary, for 2-out graphs generated with five scaling iterations, finding a maximum matching cardinality in the subgraph returns a larger cardinality than the matchings found by KS_{R1}. There is however still a gap between the best possible (red line) and what KS_{R1} 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 similar to how 2outMC operates on bipartite 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_{R1} 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 Phase-1 of KS_{R1} on One-Out 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 Figure 3.2. The ratio remains below 1% for all the graphs in our dataset except one. For this graph the ratio increases to 8%. After examination, we found that the returned matching in the 1-out subgraph is maximum for the host graph as well (i.e., the number of odd components is also large in the original graph and unavoidable).

3.2.2 On a hard synthetic instance for KS_{R1}

We compare KS_{R1} and One-Out on a family of synthetic graphs. This family extends the synthetic family $\mathcal{J}$ from Section 2.3.3 of the previous chapter and whose adjacency matrix was depicted in Figure 2.9. Recall from that section that the adjacency matrix of a graph in $\mathcal{J}$ had a full block in the upper left part which impacted negatively the performance of KS-like heuristics. In addition, the number of edges in the graphs of $\mathcal{J}$ were affected by a parameter t

		One-Out					
	KS_{R1}	5 iters.		10 iters.		20 iters.	
t	quality	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 3.2 – Results for graphs corresponding to $\mathcal{J}$ family for which KS-like algorithms struggle. The results are with $n = 5000$ and different t 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.

which impacted negatively the performance of KS_{R1} as it increased.

The extension to undirected graphs is rather straightforward. We create a new adjacency matrix $\mathbf{B}$ by using the one in Figure 2.9 and zeroing out any diagonal entries since they are not allowed in undirected graphs. Then, by similar reasoning as in the bipartite case one can see that as long as t increases the quality of KS-like heuristics drops as it becomes harder and harder to apply the deterministic rules.

Table 3.2 shows that the quality of KS_{R1} drops to 0.63 as t increases. In comparison, One-Out heuristic with five scaling iterations maintains a good quality for small t values. However, a better scaling with more iterations (10 and 20 for $t = 128$ and $t = 512$, respectively) is required to guarantee the desired matching quality—see the scaling error in the table.

3.2.3 On large-scale graphs

These experiments are performed on a machine running 64-bit CentOS 6.5, has 30 cores each of which is an Intel Xeon CPU E7D4870 v2 core operating at 2.30 GHz. To choose five large-scale matrices from the SuiteSparse Matrix collection, 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 3.3. This table also contains the run times and matching qualities of the original KS_{R1}, and the proposed One-Out and Two-Out heuristics. The proposed heuristics have five scaling iterations and also apply KS_{R1} at the end for ensuring maximality.

The run time of the proposed heuristics are analyzed in four stages in Table 3.3. The *Scale* stage scales the matrix with five iterations, the *k-out* stage chooses the neighbors and constructs the k-out subgraph, the KS_{R1} stage applies KS_{R1} on the k-out subgraph, and $KS_{R1}+$ is the stage for the additional KS_{R1} application at the end. The total time with these four stages are also shown. The quality results are consistent with the experimental results obtained on the first dataset. As the table shows, the proposed heuristics are faster than the original KS_{R1} on the input graph. For 1-out graphs, the proposed approach is 2.5–3.9 faster than KS_{R1} on the original graph. The speedups are in between 1.5–2.6 for 2-out graphs with five iterations. We also note that we did not include cost of randomization of the edges for KS_{R1} which would further slow down its performance.

3.3 Concluding remarks

We proposed heuristics for approximating the maximum cardinality matchings on general undirected graphs. The heuristics scale the adjacency matrix of a given graph, and then select a

					One-Out-S5-KS$_{R1}$+						Two-Out-S5-KS$_{R1}$+				
			KS$_{R1}$			Execution time (seconds)						Execution time (seconds)			
Matrix	$\|V\|$	$\|E\|$	*Quality*	*time*	*Quality*	*Scale*	*1-out*	KS$_{R1}$	KS$_{R1}$+	Total	*Quality*	*2-out*	KS$_{R1}$	KS$_{R1}$+	Total
`cage15`	5.2	94.0	1.00	5.82	0.93	0.67	0.85	0.65	0.05	2.21	0.99	1.48	1.78	0.01	3.94
`dielFilterV3real`	1.1	88.2	0.99	3.36	0.98	0.52	0.36	0.07	0.01	0.95	0.99	0.62	0.16	0.00	1.30
`hollywood-2009`	1.1	112.8	0.93	4.18	0.91	1.12	0.45	0.06	0.02	1.65	0.95	0.76	0.13	0.01	2.01
`nlpkkt240`	28.0	746.5	0.98	52.95	0.93	4.44	4.99	3.96	0.27	13.66	0.98	9.43	10.56	0.11	24.54
`rgg_n_2_24_s0`	16.8	265.1	0.98	19.49	0.93	2.33	2.33	2.08	0.17	6.91	0.98	4.01	6.70	0.11	13.15

Table 3.3 – Summary of the results with five large-scale matrices for original KS$_{R1}$ and the proposed One-Out and Two-Out heuristics which generate maximal matchings with extra KS$_{R1}$ and five scaling iterations. The scaling time for Two-Out is identical to the one for One-Out.

subgraph on which a maximum cardinality matching is obtained. The main algorithm One-Out works by selecting a random 1-out subgraph, while two other variants enrich this 1-out subgraph with additional edges. We showed that One-Out should yield an approximation of the maximum cardinality matching that is very close to $0.866 - \log n/n$. This bound was verified through numerous experiments. Our theoretical analysis can be perceived as an analysis of the well-known KS$_{R1}$ heuristic on the random 1-out graph model. Our experiments suggest that one of the heuristics has performance on par with KS$_{R1}$ whilst being faster and more reliable.

A more rigorous treatment and elaboration of the variants described in Section 3.1.2 seems worthwhile. Although KS$_{R1}$ works well for the generated subgraphs of these two variants, we would like to investigate further and consider whether it is possible to extend 2outMC for undirected graphs. We are also interested in adapting TruncRW for undirected graphs. While the underlying theory for TruncRW has not been shown to hold for d-regular undirected graphs, we expect that its behavior should remain good. On a more practical note, we also plan to work on the parallel implementations of the One-Out algorithm and its variants.

Chapter 4

Matchings in hypergraphs

In this chapter we investigate heuristics for the maximum cardinality matching problem in d-partite, d-uniform hypergraphs, or MAX-d-DM, which was defined in Chapter 1. The results of this chapter were published in the proceedings of the SEA^2 2019 conference [C1]. For convenience, we briefly recall some of the necessary definitions. A hypergraph $H = (V, E)$ consists of a finite set V and a collection E of subsets of V. The set V is called vertices, while the collection E is called hyperedges. A hypergraph 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. A matching in a hypergraph is a set of disjoint hyperedges. MAX-d-DM then asks for the matching with largest cardinality.

Since MAX-d-DM is NP-Complete [70], we propose five heuristics. The first two heuristics are adaptations of the GREEDY [41] and Karp–Sipser [72] heuristics which were discussed in Chapter 2. We will use GREEDYH and KARPSIPSERH for the proposed generalizations to distinguish them from their bipartite counterparts. GREEDYH traverses the hyperedge list in random order and adds a hyperedge to the matching whenever possible. KARPSIPSERH introduces certain rules to GREEDYH to improve the cardinality of the returned matching. The third heuristic KARPSIPSERHSCALING is inspired by the scaling-based approaches that were examined in Chapters 2–3. The fourth heuristic KARPSIPSERHMINDEGREE can be seen as a modification of the third one to simplify scaling and leads to reduced computation time. The last heuristic BIPARTITEREDUCTION recursively finds a matching for a reduced, $(d-1)$-dimensional problem and adapts it for d-dimensions using a weighted bipartite matching algorithm. We perform experiments to evaluate the performance of these heuristics on special classes of random hypergraphs as well as real-life data.

Another 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 [84], can be used to compute large matchings in hypergraphs. This is possible because each matching in the hypergraph corresponds to an independent set in the line graph and vice versa. This approach, although promising quality-wise, can be highly 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 many cases it is not. We have such instances in the experiments. Furthermore, due to the quadratic size of line graphs, even heuristics with linear complexity for the independent set problem can be inefficient.

The rest of the chapter is organized as follows. In Section 4.1 we disuss our heuristics for the MAX-d-DM problem and Section 4.2 presents the experimental results. Section 4.3 summarizes the chapter and discusses some potential future work.

4.1 Heuristics for maximum d-dimensional matching

Recall from Chapter 1 that a matching which cannot be extended with additional hyperedges is called maximal. We will propose heuristics for finding maximal matchings on d-partite, d-uniform hypergraphs. In such hypergraphs any maximal matching is a d-approximate matching. The bound is tight and can be verified for $d = 3$ as follows. Let H be a 3-partite $3 \times 3 \times 3$ hypergraph with the following hyperedges $e_1 = (1, 1, 1), e_2 = (2, 2, 2), e_3 = (3, 3, 3)$ and $e_4 = (1, 2, 3)$. The maximum matching is $\{e_1, e_2, e_3\}$ but the hyperedge $\{e_4\}$ alone forms a maximal matching. The hypergraphs to show approximation tightness for higher values of d are defined in a similar manner.

4.1.1 A Greedy heuristic for Max-d-DM

As discussed in Chapter 2 there exist two main variants of the GREEDY algorithm for the graph case. The first one iterates over the edges [41], while the other iterates over the vertices [104]. We adapt the first variant to our problem and call it GREEDYH. It traverses the hyperedges in random order and adds the current hyperedge to the matching whenever possible. Since any maximal matching is possible as its output, GREEDYH is a d-approximation heuristic. It provides matchings of varying quality, depending upon the order in which the hyperedges are processed.

4.1.2 KarpSipserH for Max-d-DM

A detailed description of the matching heuristic due to Karp and Sipser [72] can be found in Chapter 2. We now propose its adaptation for d-partite, d-uniform hypergraphs. Similar to the original version for graphs, the adaptation for hypergraphs iteratively adds a random hyperedge to the matching, removes its d endpoints, as well as their hyperedges. However, the random selection is not applied whenever hyperedges defined by the following lemmas appear.

Lemma 4.1. *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, then 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 is not valid.

Lemma 4.2. *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 maximum, u_j must be matched by M' (otherwise, M' could 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 4.2 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 4.2 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 remains d-partite and d-uniform).

Although the extended rules usually lead to improved results in comparison to GreedyH, KarpSipserH still adheres to the d-approximation bound of maximal matchings. To see this, we can use the 3-dimensional toy example given as a worst-case for GreedyH at the beginning of Section 4.1. There, KarpSipserH 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.

4.1.3 KarpSipserHScaling for Max-d-DM

KarpSipserH can be modified for better decisions in case neither of the two rules hold. In our variant, instead of a random selection, we first scale the adjacency tensor of the given hypergraph H and obtain an approximate d-stochastic tensor $\mathbf{T}$. We then augment the matching by adding the hyperedge which corresponds to the largest value in $\mathbf{T}$. The modified heuristic is summarized in Algorithm 4.1.

Our inspiration comes from the $d = 2$ case and more specifically from the relation between scaling and matching which was discussed in the preceding chapters. 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

Algorithm 4.1: KarpSipserHScaling: The scaling-based extension of Karp–Sipser in hypergraphs

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

```
M ← ∅ {Initially M is empty}
S ← ∅ {Stack for the merges for Rule-2}
while H is not empty do
    Remove the isolated vertices from H
    if ∃e = (u_1, ..., u_d) as in Rule-1 then
        M ← M ∪ {e} {Add e to the matching}
        Apply the reduction for Rule-1 on H
    else if ∃e = (u_1, ..., u_d), e' = (u'_1, ..., u'_d) and J as in Rule-2 then
        Let j be the part index where j ∉ J
        Apply the reduction for Rule-2 on H by introducing the vertex u_j u'_j
        E' ← {(v_1, ..., u_j u'_j, ..., v_d) : for all (v_1, ..., u_j, ..., v_d) ∈ E}
          {memorize the hyperedges of u_j}
        S.push(e, e', u_j u'_j, E') {Store the current merge}
    else
        T ← Scale(adj(H)) {Scale the adjacency tensor of H}
        e ← arg max_(u_1,...,u_d) (T_(u_1,...,u_d)) {Find the maximum entry in T}
        M ← M ∪ {e} {Add e to the matching}
        Remove all hyperedges of u_1, ..., u_d from E
        V ← V \ {u_1, ..., u_d}
while S ≠ ∅ do
    (e, e', u_j u'_j, E') ← S.pop() {Get the most recent merge}
    if u_j u'_j is not matched by M then
        M ← M ∪ {e}
    else
        Let e'' ∈ M be the hyperedge matching u_j u'_j
        if e'' ∈ E' then
            Replace u_j u'_j in e'' with u'_j
            M ← M ∪ {e'}
        else
            Replace u_j u'_j in e'' with u_j
            M ← M ∪ {e}
```

chosen. Unfortunately however for $d \geq 3$, there is no equivalent of Birkhoff's decomposition theorem (see Section 1.3.2) as demonstrated by the following lemma.

Lemma 4.3. *For $d \geq 3$, there exist extreme points in the set of d-stochastic tensors which are not permutation tensors.*

Proof. We provide a 3-stochastic $2 \times 2 \times 2$ tensor $\mathbf{T}^3$ with an inspiration from [26]. For convenience, we depict $\mathbf{T}^3$ by two 2×2 matrices which are the marginals of the 3rd dimension:

$$\mathbf{T}^3_{:,:,1} = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \text{ and } \mathbf{T}^3_{:,:,2} = \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{pmatrix} .$$

The maximum matching cardinality in the hypergraph defined by the tensor is 1. Furthermore, $\mathbf{T}^3$ cannot be written as a linear combination of permutation tensors. This particular extreme point can be extended for higher d by setting $\mathbf{T}^d_{u_1,u_2,u_3,\ldots,u_3} = \mathbf{T}^3_{u_1,u_2,u_3}$ for each nonzero element

$\mathbf{T}^3_{u_1,u_2,u_3}$ and for higher n by setting $\mathbf{T}^d{}_{3,\ldots,3} = \cdots = \mathbf{T}^d{}_{n,\ldots,n} = 1$. □

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 KarpSipserH described above, the scaling indeed helps the entries corresponding to e_4, e_5 and e_6 to become zero.

Let $\mathbf{S}^3$ be the tensor obtained by swapping the 2nd and 3rd dimensions of $\mathbf{T}^3$. We can see that the tensor $\frac{1}{2}\mathbf{T}^3 + \frac{1}{2}\mathbf{S}^3$ has a perfect matching, however, obtained by a linear combination of two extreme points that are not permutation tensors. This shows that even if we select an entry from an extreme point without a perfect matching, we do not necessarily reduce our chances of obtaining a good matching as entries outside that extreme point also exist.

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) like in the graph case. 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 as was the case with graphs.

4.1.4 Hypergraph matching via pseudo scaling

In Algorithm 4.1, 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 mimic the first iteration of the Sinkhorn–Knopp algorithm and use the inverse of the degree of a vertex as its scaling vector. This avoids 10–20 iterations of Sinkhorn–Knopp and the $\mathcal{O}(|E|)$ cost for each. However, as can be understood, the scaling is not exact. With this approach each hyperedge $\{i_1, \ldots, i_d\}$ is associated with a value $\frac{1}{\prod_{j=1}^{d} \deg_{i_j}}$. The selection procedure then is the same as that of Algorithm 4.1, i.e., the edge with the maximum value is added to the matching set. We refer to this algorithm as KarpSipserHMindegree, as it selects a hyperedge based on a function of the degrees of the vertices.

4.1.5 Reduction to bipartite graph matching

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. Aharoni and Haxell [2] investigated matchability in $(d-1)$-dimensional subhypergraphs to provide an equivalent of Hall's Theorem for d-partite hypergraphs. These observations lead us to propose a heuristic called BipartiteReduction. 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 $H = (V, E)$ where $V = V_1 \cup V_2 \cup V_3$. 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

$$w(u, v) = |\{z : (u, v, z) \in E\}| \,. \tag{4.1}$$

In this case, a maximum weighted (product, sum, etc.) matching algorithm can be used to obtain a matching M_G on G. Otherwise if the edges are not weighted, one can simply use a maximum cardinality matching algorithm. 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 (4.1) is used, then the following proposition holds.

Proposition 4.1. *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.*

Proof. Consider a node $u \in V_1$ and let it be matched with $v \in V_2$ in M_G. The number of edges involving uv in G' is $|\{z : (u, v, z) \in E\}|$. We see that this number is equivalent to $w(u, v)$, and the result follows by treating each matched pair in M_G in this way. □

Proposition 4.1 shows that by finding maximum weighted matching M_G, we guarantee that the largest possible number of edges will survive in G'.

The process is similar for d-dimensional matching. 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, BipartiteReduction does not have any approximation guarantee. We depict this with the following lemma.

Lemma 4.4. *If algorithms for the maximum cardinality or the maximum weighted matching (with the suggested edge weights (4.1)) problems are used, then* BipartiteReduction *has a worst-case approximation ratio of $\Omega(n)$.*

Proof. We discuss initially the case for $d = 3$ and assume $n \geq 5$. Consider an $n \times n \times n$ hypergraph H with edges $e_i = (u_i, v_i, z_i), e'_i = (u_i, v_{1+i \mod n}, z_2)$ and $e''_i = (u_i, v_{1+i \mod n}, z_3)$ for $i \in \{1, \ldots, n\}$. There is a perfect matching containing all edges $e_1, \ldots, e_n$.

Suppose we create G by projecting the 3rd dimension. Then, the edges in G are either of the form $h_i = (u_i, v_i)$ with $w(h_i) = 1$ or $h'_i = (u_i, v_{1+i \mod n})$ with $w(h'_i) = 2$. Both $\{h_1, \ldots, h_n\}$ and $\{h'_1, \ldots, h'_n\}$ form perfect matchings in G. If the weight function (4.1) is used, the algorithm will necessarily find the perfect matching $\{h'_1, \ldots, h'_n\}$. Otherwise, any matching algorithm can arbitrarily return $\{h'_1, \ldots, h'_n\}$.

Assuming that $\{h'_1, \ldots, h'_n\}$ is returned, the graph G' will have $2n$ edges. The edges will be either in the form $he_i = (u_i v_{1+i \mod n}, z_2)$ or $he'_i = (u_i v_{1+i \mod n}, z_3)$ for $i \in \{1, \ldots, n\}$. As seen, z_2 and z_3 are the only two vertices of the 3rd dimension which can be matched.

The algorithm will return a perfect matching, if we project a dimension other than the 3rd one. To extend H such that the approximation ratio is $\Omega(n)$ whichever dimension is projected, we need to introduce the following four additional set of edges: $e_i^{(3)} = (u_2, v_i, z_{1+i \mod n}), e_i^{(4)} = (u_3, v_i, z_{1+i \mod n}), e_i^{(5)} = (u_{1+i \mod n}, v_2, z_i)$ and $e_i^{(6)} = (u_{1+i \mod n}, v_3, z_i)$ for $i \in \{1, \ldots, n\}$ that mirror $\{e'_1, \ldots e'_n\}$ and $\{e''_1, \ldots, e''_n\}$. In this case, the maximum matching in G' will always be 5, as again the edges in $\{e_1, \ldots, e_n\}$ will be ignored.

The result holds for higher d by noting that H alongside its extension are valid 3-partite hypergraphs that can occur after a matching for vertices in dimensions $i_1, \ldots, i_{d-2}$ has been found. □

4.1.6 Performing local search

A local search heuristic was proposed by Hurkens and Schrijver [64]. 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 [27] 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.

4.2 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 compare the adapted GreedyH and KarpSipserH heuristics with the proposed KarpSipserHScaling and KarpSipserHMindegree heuristics. For $d = 3$, we also consider LocalSearch [64], 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 GreedyH and KarpSipserH with different random decisions and take the maximum cardinality obtained. Since KarpSipserHScaling or KarpSipserHMindegree do not pick hyperedges randomly, we run them only once. We perform 20 steps of the scaling procedure in KarpSipserHScaling. 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.

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

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

Here, we consider and experiment with the model of random k-out, d-partite, d-uniform hypergraphs described in Section 1.2.

We first investigate the existence of perfect matchings in random k-out, d-partite, d-uniform hypergraphs. For this purpose, we implemented the linear program of d-dimensional matching in the CPLEX solver [1]. We experimented 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 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 4.1 – 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.

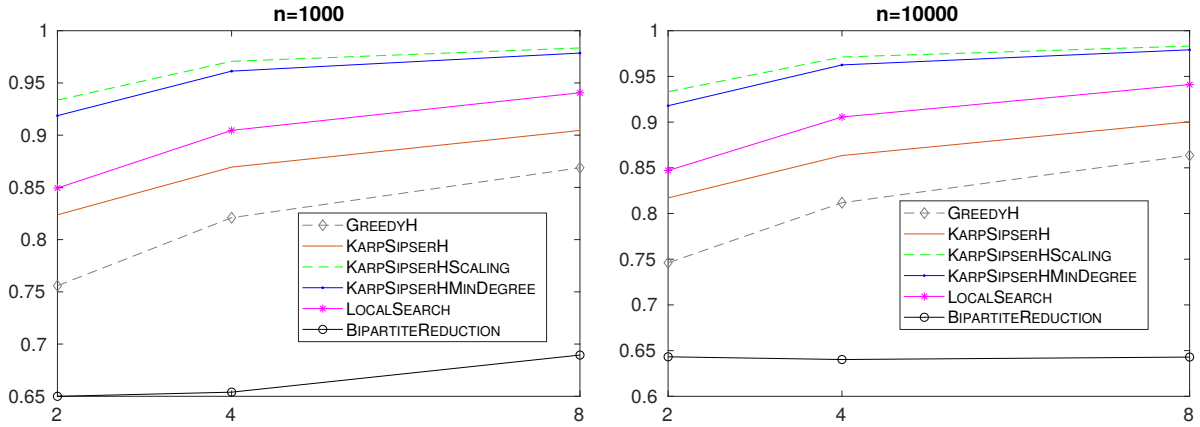
hypergraphs and computed their maximum cardinality matchings using CPLEX. For $k = d^{d-3}$, we encountered several hypergraphs with no perfect matching, especially for $d = 3$. The hypergraphs with $k = d^{d-2}$ were also lacking a perfect matching for $d = 2$. However, all the hypergraphs we created 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 4.1. 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 being equal to powers of 2 for as long as $k \leq d \log d$. The results are summarized in Figure 4.1. 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.

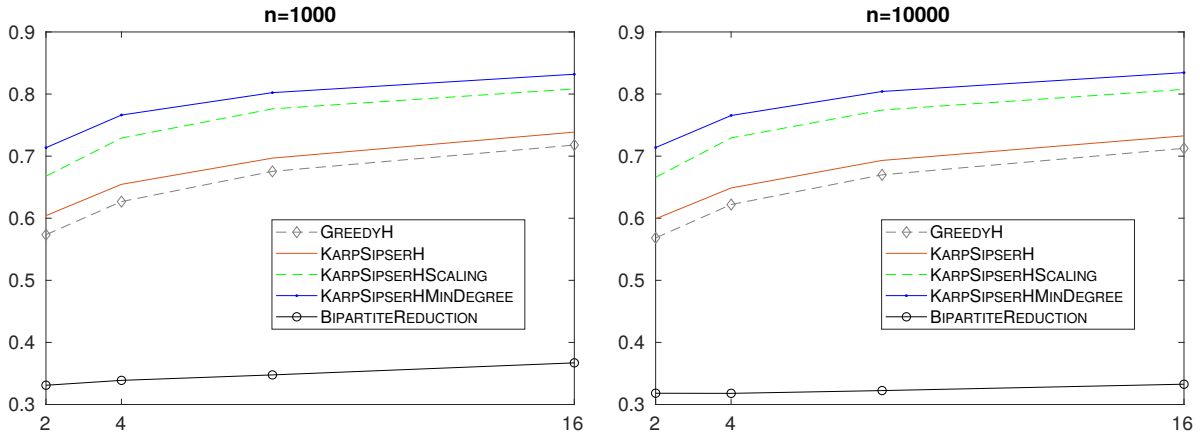
As seen, KarpSipserHScaling and KarpSipserHMindegree have the best performance, comfortably beating the other alternatives. For $d = 3$, KarpSipserHScaling has better performance than KarpSipserHMindegree, but when $d > 3$ we see that KarpSipserHMindegree has the best performance. KarpSipserH performs better than GreedyH. However, their performances get closer as d increases. This is due to the fact that the conditions in Lemmas 4.1 and 4.2 have more restrictions and are applied less frequently as d increases. As a consequence, KarpSipserH mimics partly the behavior of GreedyH. BipartiteReduction 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.

4.2.1.2 Sparse random d-partite, d-uniform hypergraphs

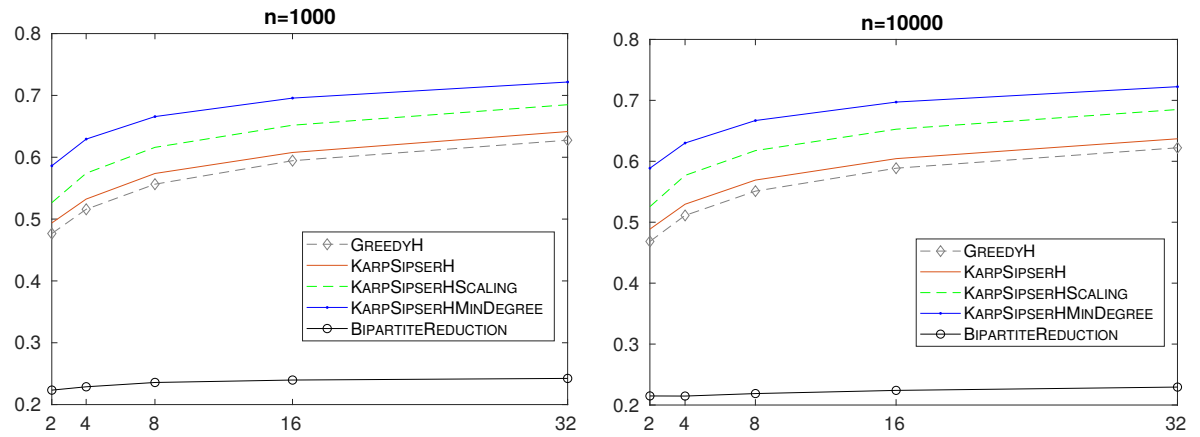
Here, we consider a random d-partite, d-uniform hypergraph H_i is created with $i \times n$ hyperedges. The parameters used for this experiment are $i \in \{1, 3, 5, 7\}$, $n \in \{4000, 8000\}$, and $d \in \{3, 6, 9\}$. Each H_i is created by choosing the vertices of a hyperedge uniformly at random 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. We again generate ten random instances for each parameter setting. We do not present results for BipartiteReduction as it was always



(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 4.1 – The performance (i.e., cardinality over n) 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 LocalSearch for $d = 6$ and $d = 9$.

	H_i: Hypergraph										H_{i+M}: Hypergraph + perfect matching									
			Local		KarpSipserH								Local		KarpSipserH					
	GreedyH		Search		Default		Scaling		MinDegree		GreedyH		Search		Default		Scaling		MinDegree	
i	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000	4000	8000
1	0.43	0.42	0.47	0.47	0.49	0.48	0.49	0.48	0.49	0.48	0.75	0.75	0.93	0.93	1.00	1.00	1.00	1.00	1.00	1.00
3	0.63	0.63	0.71	0.71	0.73	0.72	0.76	0.76	0.78	0.77	0.72	0.71	0.82	0.81	0.81	0.81	0.99	0.99	0.92	0.92
5	0.70	0.70	0.80	0.80	0.78	0.78	0.86	0.86	0.88	0.88	0.75	0.74	0.84	0.84	0.82	0.82	0.94	0.94	0.92	0.92
7	0.75	0.75	0.84	0.84	0.81	0.81	0.94	0.94	0.93	0.93	0.77	0.77	0.87	0.87	0.83	0.83	0.96	0.96	0.94	0.94

(a) $d = 3$, without (left) and with (right) the planted matching.

	H_i: Hypergraph								H_{i+M}: Hypergraph + perfect matching							
			KarpSipserH								KarpSipserH					
	GreedyH		Default		Scaling		MinDegree		GreedyH		Default		Scaling		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

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

	H_i: Hypergraph								H_{i+M}: Hypergraph + perfect matching							
			KarpSipserH								KarpSipserH					
	GreedyH		Default		Scaling		MinDegree		GreedyH		Default		Scaling		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

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

Figure 4.2 – Performance (i.e., cardinality over n) 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. . Default refers to KarpSipserH from Section 4.1.2.

worse than the others, as before.

The average quality of different heuristics on these instances is shown in Figure 4.2. These experiments confirm that KarpSipserH performs consistently better than GreedyH. Furthermore, KarpSipserHScaling performs significantly better than KarpSipserH. KarpSipserHScaling works even better than the local search heuristic, and it is the only heuristic that is capable of finding planted perfect matchings for a significant number of the runs. In particular when $d > 3$, it finds a perfect matching on H_{i+M}'s in all cases except for when $d = 6$ and $i = 7$. For $d = 3$, it finds a perfect matching only when $i = 1$ and attains a near perfect matching when $i = 3$. Interestingly KarpSipserHMindegree outperforms KarpSipserHScaling on H_is but is dominated on H_{i+M}s, where it is the second best performing heuristic.

4.2.2 On synthetic hypergraphs

Here, we evaluate the use of scaling and the importance of Rule-1 and Rule-2 in two different families of hypergraphs. These hypergraphs generalize the two synthetic families of graphs $\mathcal{I}$ and $\mathcal{J}$ seen in Chapter 2.

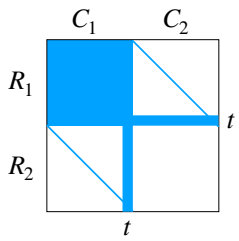


Figure 4.3 – $\mathbf{A}_{\mathcal{J}}$: Adjacency matrix representation for the synthetic family $\mathcal{J}$.

t	GreedyH	Local Search	KarpSipserH Default	KarpSipserH Scaling	KarpSipserH 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 4.2 – Matching quality of the proposed heuristics on 3-partite, 3-uniform hypergraphs corresponding to $\mathbf{T}_{\mathcal{J}}$ with $n = 300$ vertices in each part. Default refers to KarpSipserH from Section 4.1.2.

4.2.2.1 Scaling vs no scaling

To evaluate and emphasize the contribution of scaling better, we compare the performance of the heuristics on a particular family of d-partite, d-uniform hypergraphs where their bipartite counterpart $\mathcal{J}$ was used in Chapter 2. Let $\mathbf{A}_{\mathcal{J}}$ be an $n \times n$ matrix described in accordance with the synthetic family $\mathcal{J}$ of Section 2.3.3 and Figure 2.9 and shown again in Figure 4.3 for convenience. Recall that as the parameter t grew larger, Karp–Sipser had trouble applying deterministic rules and its quality dropped. To adapt this family to hypergraphs/tensors, we generate a 3-dimensional tensor $\mathbf{T}_{\mathcal{J}}$ such that the nonzero pattern of each marginal of the 3rd dimension is identical to that of $\mathbf{A}_{\mathcal{J}}$, i.e., $\mathbf{T}_{\mathcal{J}}(:,:,i) = \mathbf{A}_{\mathcal{J}}$ for $i \leq n$.

Table 4.2 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 KarpSipserHScaling heuristic always finds the perfect matching as does KarpSipserHMindegree. However, both GreedyH and KarpSipserH perform significantly worse. Furthermore, LocalSearch returns a 0.99-approximation in every case because it ends up in a local optima.

4.2.2.2 Rule-1 vs Rule-2

We finish the discussion on the synthetic data by focusing on KarpSipserH.

We present a family of hypergraphs to demonstrate that, similar to the graph case, Rule-2 can lead to better performance than using Rule-1 only. This family extends the family $\mathcal{I}$ from Section 2.3.3. We use KarpSipserH$_{R1}$ to refer to KarpSipserH without Rule-2.

As before, we start from the bipartite case. Let $\mathbf{A}_{\mathcal{I}}$ be a $n \times n$ matrix described according to the description of family $\mathcal{I}$. Recall that the adjacency matrix of graphs from the $\mathcal{I}$ family had an upper triangular matrix and two additional nonzero entries $(2, 1)$ and $(n, n\text{-}1)$. To extend the graph of $\mathbf{A}_{\mathcal{I}}$ to a hypergraph $H_{\mathcal{I}}$, we proceed as follows. Let $\mathbf{T}_{\mathcal{I}}$ be a d-dimensional $n \times \cdots \times n$ tensor which will represent the adjacency tensor representation of $H_{\mathcal{I}}$. We set

	d			
	3		6	
n	quality	r/n	quality	r/n
1000	0.85	0.47	0.80	0.31
2000	0.87	0.56	0.80	0.30
4000	0.75	0.17	0.84	0.45

Table 4.3 – Quality of matching and the number r of the applications of Rule-1 over n in KarpSipserH$_{R1}$, for hypergraphs corresponding to $\mathbf{T}_{\mathcal{J}}$. KarpSipserH obtains perfect matchings.

$(\mathbf{T}_{\mathcal{J}})_{i,j,\ldots,j} = 1$ for $1 \leq i \leq j \leq n$ and $(\mathbf{T}_{\mathcal{J}})_{1,2,\ldots,2} = (\mathbf{T}_{\mathcal{J}})_{n,n-1,\ldots,n-1} = 1$. By similar reasoning as in the bipartite case, we see that KarpSipserH with both reduction rules will obtain a perfect matching by applying Rule-2 and merging vertices $1, 2$ of the 3rd dimension as a first step whereas KarpSipserH$_{R1}$ will struggle. We give some results in Table 4.3 that show the difference between the two. We test for $n \in \{1000, 2000, 4000\}$ and $d \in \{3, 6\}$, and show the quality of KarpSipserH$_{R1}$ and the number of times that Rule-1 is applied over n. We present the best result over 10 runs. As seen in Table 4.3, KarpSipserH$_{R1}$ obtains matchings that are about 13–25% worse than KarpSipserH. Furthermore, the larger the number of Rule-1 applications is, the higher the quality is.

4.2.3 On real-life hypergraphs

We also evaluate the performance of the proposed heuristics on some real-life tensors selected from the FROSTT library [102]. The descriptions of the tensors are given in Table 4.4. 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 4.4 do not have an equal number of vertices. In this case, the scaling algorithm works among the same lines as it does on rectangular matrices. Let $n_i = |V_i|$ be the cardinality at ith dimension and $n_{max} = \max_{1 \leq i \leq d} n_i$ be the maximum one. By slightly modifying Sinkhorn–Knopp, for each iteration of KarpSipserHScaling, 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 4.4 resemble those from previous sections; KarpSipserHScaling has the best performance and is slightly superior to KarpSipserHMindegree. GreedyH and KarpSipserH are close to each other and when it is feasible, LocalSearch is better than them. In addition we see that in these instances BipartiteReduction exhibits a good performance: its performance is at least as good as KarpSipserHScaling for the first three instances, but about 10% worse for the last one.

4.2.4 Comparison with an independent set solver

We compare KarpSipserHScaling and KarpSipserHMindegree with the idea of reducing Max-d-DM to the problem of finding an independent set in the line graph of the given hypergraph. We show that this transformation can lead to good results, but is restricted because line graphs can require too much space.

Tensor	d	Dimensions	Hyperedges	GreedyH	Local Search	KarpSipserH Default	KarpSipserH Mindegree	KarpSipserH Scaling	Bipartite Reduction
`Uber` [102]	3	$183 \times 1140 \times 1717$	1.11	183	183	183	183	183	183
`nips` [53]	3	$2482 \times 2862 \times 14036$	3.10	1847	1991	1839	2005	2007	2007
`Nell-2` [19]	3	$12092 \times 9184 \times 28818$	76.88	3913	4987	3935	5100	5154	5175
`Enron` [100]	4	$6,066 \times 5699 \times 244268 \times 1176$	54.20	875	-	875	988	1001	898

Table 4.4 – Four real-life tensors and the matching cardinality found by the proposed heuristics on the corresponding hypergraphs. The number of hyperedges is in the order of millions. No result for LocalSearch for `Enron`, as it is four dimensional. Default refers to KarpSipserH from Section 4.1.2.

Hypergraph	KaMIS Line graph gen. time	KaMIS Round 1 quality	KaMIS Round 1 time	KaMIS Output quality	KaMIS Output time	GreedyH quality	KarpSipserH Scaling quality	KarpSipserH Scaling time	KarpSipserH Mindegree quality	KarpSipserH Mindegree 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
H_3, $n = 8000$, $d = 3$	1	0.77	16	0.81	602	0.63	0.76	5	0.77	1
H_{3+M}, $n = 8000$, $d = 3$	2	0.89	25	1.00	430	0.70	1.00	11	0.91	1

Table 4.5 – Run time (in seconds) and performance (i.e., cardinality over n) comparisons between KaMIS, GreedyH,KarpSipserHScaling, and KarpSipserHMindegree. The time required to create the line graphs should be added to KaMIS's overall time.

We use KaMIS [84] 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, GreedyH, KarpSipserHScaling, and KarpSipserHMindegree on a few hypergraphs from previous tests. The results are summarized in Table 4.5. The run time of GreedyH 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 KarpSipserHScaling and KarpSipserHMindegree, it is also significantly slower (its principle is to deliver high quality results). We also observe that the pseudo scaling of KarpSipserHMindegree indeed helps to reduce the run time compared to KarpSipserHScaling.

The line graphs of the real-life instances from Table 4.4 are too large to be handled. We estimate using known techniques [24] the number of edges in these graphs to range from 1.5×10^{10} to 4.7×10^{13}. The memory needed ranges from 126GB to 380TB if edges are stored twice (assuming 4 bytes per edge).

4.3 Concluding remarks

We proposed heuristics for the Max-d-DM problem by generalizing existing heuristics for the maximum cardinality matching in bipartite graphs. The experimental analysis on various hy-

pergraphs/tensors showed the effectiveness and efficiency of the proposed heuristics. As future work, we plan to investigate the stated conjecture that d^{d-1}-out random hypergraphs have perfect matchings almost always, and further analyze the theoretical guarantees of the proposed algorithms. Another interesting direction for future work is examining the weighted version of the MAX-d-DM problem. In this version, the hyperedges have weights, and one looks for a matching whose hyperedges have the maximum sum of weights.

Chapter 5

Counting the number of perfect matchings in graphs

In this chapter, we investigate efficient randomized methods for approximating the number of perfect matchings in bipartite graphs and general undirected graphs. The results of this chapter are published in the Discrete Applied Mathematics journal [J2]. For convenience, we recall some basic definitions from Chapter 1. The permanent of an $n \times n$ square matrix $\mathbf{A}$ is defined as $Per(\mathbf{A}) = \sum_{\sigma} \prod_i a_{i,\sigma(i)}$, where the summation runs over all permutations σ of $1, \ldots, n$. The value of the permanent in the adjacency matrix $\mathbf{A}_G$ of a bipartite graph G is equal to the number of perfect matchings in G. A similar function to count the number of perfect matchings in general undirected graphs from their adjacency matrix representation can also be defined. Calculating the value of the permanent exactly is #P-Complete [106], where #P is the complexity class containing the counting problems that are associated with the decision problems from the complexity class NP.

Rasmussen [99] proposed a practically efficient way of estimating the permanent of a 0-1 matrix, or the number of perfect matchings in bipartite graphs. This initial work was followed by a series of others [8, 48, 49]. Rasmussen's iterative algorithm chooses uniformly and at random a nonzero from the first row in the matrix at hand and discards the first row and the column of the chosen nonzero. This step is repeated until the whole matrix is consumed or there is an empty row remaining. In the former case, an estimate of a nonzero value is returned which is based on the number of nonzeros of the first rows at each step; in the latter case, zero is returned as an estimate.

Similar to the previous chapters, our main tool will be a sparse matrix scaling algorithm. We apply matrix scaling on the adjacency matrix of the given graph (bipartite or general) to assign probabilities on the edges and base random choices on these probabilities. More specifically, we investigate an improvement of Rasmussen's original approach in which edges are selected non-uniformly based on a matrix scaling method, which scales a matrix to be doubly stochastic. We provide an analysis of the stated approach by meticulously tracking the steps taken. The analysis upper bounds the number of repetitions we must run to have quality guarantees on the estimated permanent. To the best of our knowledge, for scaling-based permanent estimation, this work presents the most thorough analysis in the literature with computable bounds. These ideas are also extended to counting the number of perfect matchings in graphs.

On the practical side, we present an extensive set of experiments on a variety of random, constructed, and real-life instances, some with known and others with unknown number of perfect matchings. Our experiments show that the scaling-based selection mechanism exhibits a

significantly better performance on all instances compared to the known methods. Furthermore, the practical performance of our algorithm seems to be even superior to the expected performance derived from our theoretical analysis, which implies that there exist further avenues to explore.

The rest of the chapter is organized as follows. Section 5.1 provides some theoretical background, and Section 5.2 discusses the related work in detail. Section 5.3 introduces the proposed approach and presents the pseudocode of the main algorithm for bipartite graphs. The extension to the general, undirected graphs case is presented in Section 5.4. Section 5.5 presents the experimental results and a comparison with existing algorithms. Section 5.6 summarizes the chapter and discusses some potential future work.

5.1 Theoretical background

Consider a certain quantity P that we want to measure. In our case, P is the permanent or the number of perfect matchings. Assume that we access to an unbiased estimator X such that $\mathbb{E}[X] = P$. A technique to measure P based on X is achieved by combining the output of N copies of X. More specifically, assuming X_i denotes the outcome of the ith estimator, $X' = \sum_{i=1}^{N} \frac{X_i}{N}$ is the combined estimate for P. X' then achieves an (ε, δ)-approximation whenever $\Pr(|X' - P| \leq \varepsilon P) \geq 1 - \delta$. In general, an (ε, δ)-approximation can be achieved by simulating $O\left(\frac{\mathbb{E}[X^2]}{\mathbb{E}[X]^2} \cdot \frac{1}{\varepsilon^2} \cdot \log(\frac{1}{\delta})\right)$ copies of X. For this reason, the fraction $\frac{\mathbb{E}[X^2]}{\mathbb{E}[X]^2}$ is called the critical ratio, as it is the key factor in determining whether the approximation scheme runs in polynomial time or not.

If an $n \times n$ nonnegative matrix $\mathbf{A}$ is scalable to a doubly stochastic matrix $\mathbf{S} = \mathbf{RAC}$ with positive diagonal matrices $\mathbf{R}$ and $\mathbf{C}$ then the function

$$g_{\mathbf{A}}(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T \mathbf{A} \mathbf{y} - \sum_{i=1}^{n} \ln x_i - \sum_{j=1}^{n} \ln y_j \tag{5.1}$$

attains its minimum value for positive x_i and y_i at $\mathbf{x} = \mathrm{diag}(\mathbf{R})$ and $\mathbf{y} = \mathrm{diag}(\mathbf{C})$ [67, Proposition 2]. In particular, for an $n \times n$ 0-1 matrix $\mathbf{A}$, we have $n \leq g_{\mathbf{A}}(\mathbf{x}, \mathbf{y}) \leq n + n \ln n$, where the lower bound is met by a permutation matrix, and the upper bound is met by the matrix of ones; in both cases $\mathbf{x}^T \mathbf{A} \mathbf{y} = n$.

5.2 Related work

Rasmussen [99] proposed a practically efficient randomized approach for estimating the permanent of a 0-1 matrix, or for counting perfect matchings in bipartite graphs, by randomly sampling one of the permutations in the matrix. Rasmussen's algorithm works along the following lines. It visits the first row of a matrix and among the nonzeros, chooses one with uniform probability. The first row and the column of the selected nonzero are then removed from the matrix and a smaller matrix is obtained. This process is repeated until either there exists an empty row, in which case zero is returned; or all rows of the original matrix were able to choose a column, in which case a nonzero estimator of the permanent X_{Ra} is returned. This estimator is the product of the number of nonzeros of the first rows for which the random selections are done. While in practice Rasmussen's estimator returns zero in most of the cases, it is an unbiased estimator, and its expected value is equal to the permanent. This is true because for each permutation in

the matrix the returned value of X_{Ra} is equal to the inverse of the probability of its selection. Rasmussen additionally showed that $\mathbb{E}[X_{Ra}^2] \leq Per(\mathbf{A})^2 n!$, for an $n \times n$ matrix $\mathbf{A}$, although one can reduce $Per(\mathbf{A})^2$ to $Per(\mathbf{A})$. As stated in Section 5.1, the value of $\mathbb{E}[X_{Ra}^2]$ is important, since it expresses the number of samples required to ensure approximation guarantees. Our estimator $X_{\mathbf{A}}$ works along the same lines. It also generates a perfect matching step-by-step, but the uniform selection of columns is replaced with a more effective weighted sampling mechanism. The weights used in sampling are obtained with a numerical algorithm used for scaling matrices to a doubly stochastic form.

Beichl and Sullivan [8] also investigated the same mechanism as us and performed some preliminary analysis. Their bound for $\mathbb{E}[X_{\mathbf{A}}^2]$ uses a notation which denotes the average value of the selection probabilities of all perfect matchings. Their analysis is valuable in that it shows that the scaling helps, but it does not yield computable bounds. We follow up on their work by providing some new theoretical insights as well as a more detailed experimental analysis. More specifically, we provide an analysis (with the main result in Theorem 5.2) which yields efficiently computable bounds on $\mathbb{E}[X_{\mathbf{A}}^2]$ depending on the scaling factors $\mathbf{R}$ and $\mathbf{C}$ for $\mathbf{A}$.

Fürer and Kasiviswanathan [49] discussed three randomized algorithms, called Simple, REP, and GreedyMtc for estimating the number of perfect matchings in general undirected graphs. Simple is a direct adaptation of Rasmussen's algorithm for graphs, and selects neighbors with uniform probability. REP extends Simple such that at certain points during the execution, it creates a number of copies where each copy selects its own neighbor, and the procedure continues in a similar way in each copy. The results of each copy are later combined to yield a single result. They have also discussed a similar algorithm for the bipartite case [48]. GreedyMtc attempts to assign probabilities in a better way by selecting each node with probability inversely proportional to its degree minus one. Similarly, GreedyMtc's returned estimate for the number of matchings is equal to the inverse of the product of the probabilities of the selected vertices at each step. Fürer and Kasiviswanathan concluded that Simple is easy to analyze but has high worst case bound; GreedyMtc looks good on many graphs but is difficult to analyze; and REP has the best theoretical guarantees for its performance on random (Erdös-Renyi) graphs, although its worst-case bounds on other graphs can be high. Our scaling-based can be seen as a more sophisticated variant of GreedyMtc. If GreedyMtc were to choose a random neighbor with probability equal to the degree of the neighbor, then it would have been equivalent to applying a single step of Sinkhorn–Knopp. However, the analysis of our variant is simpler and reveals more insights thanks to the global minimization properties of the doubly stochastic scaling, see the function (5.1) associated with doubly stochastic scaling.

5.3 The proposed algorithm and its analysis

Before discussing the estScalingPerm algorithm, we first motivate the use of scaling in estimating the value of the permanent. We remind from Section 1.3 that $\mathbf{A}_{ij}$ denotes the submatrix of matrix $\mathbf{A}$ that is obtained by deleting the ith row and the jth column. The following lemma highlights the close connection of the scaling factors and the permanents of a matrix and its submatrices.

Lemma 5.1. *Let $\mathbf{A}$ be an $n \times n$ 0-1 matrix with total support. Let $\mathbf{R}$ and $\mathbf{C}$ be the diagonal*

scaling matrices such that $\mathbf{S} = \mathbf{RAC}$ *is a double stochastic matrix Then,*

$$Per(\mathbf{S}) = Per(\mathbf{A}) \cdot \prod_{i=1}^{n} r_i \cdot c_i.$$

In addition,

$$r_i \cdot c_j = \frac{Per(\mathbf{A}_{ij})}{Per(\mathbf{A})} \cdot \frac{Per(\mathbf{S})}{Per(\mathbf{S}_{ij})}.$$

Proof. Since $\mathbf{R}$ and $\mathbf{C}$ are diagonal matrices, we have $Per(\mathbf{S}) = Per(\mathbf{A}) \prod r_i \prod c_j$. The equality $Per(\mathbf{S}_{ij}) = Per(\mathbf{A}_{ij}) \prod_{k \neq i} r_k \prod_{\ell \neq j} c_\ell$ holds, as all diagonal products in $\mathbf{S}_{ij}$ have the same value $\prod_{k \neq i} r_k \prod_{\ell \neq j} c_\ell$. Dividing the two equalities side by side yields the second result. □

5.3.1 The algorithm

The proposed algorithm to estimate the permanent is shown in Algorithm 5.1. The algorithm takes an $n \times n$, 0-1 matrix $\mathbf{A}$ with a nonzero permanent and produces a random variable denoted as $X_{\mathbf{A}}$ as well as a perfect matching (this is for the analysis). Initially $X_{\mathbf{A}}$ is equal to one. The algorithm proceeds in n steps. At every step, the algorithm adds a nonzero entry to a matching, thereby obtaining a perfect matching at the end. At step i, a nonzero entry in the ith row of $\mathbf{A}$ is chosen among those columns $\mathbf{A}$ which have not been matched yet. The nonzero entry chosen at row i defines the matched column. Since the ith row is the first row at step i, we use $\sigma_1^{(i)}$ to denote the column chosen for i, and $\mathbf{A}^{(i)}$ to denote the remaining matrix. The nonzeros are selected according to the values of the entries in a doubly stochastically scaled version of the remaining matrix. The random variable $X_{\mathbf{A}}$ is multiplied by the reciprocal of the value of the chosen nonzero. For the algorithm to work, we discard the nonzeros in $\mathbf{A}^{(i)}$ that cannot be put into a perfect matching. This is achieved by applying the Dulmage–Mendelsohn [40, 97] decomposition, and filtering out the entries that fall into the off-diagonal blocks in a fine decomposition [97].

We first comment on the run time complexity of the algorithm. There are n steps, and each step requires computing a Dulmage–Mendelsohn decomposition of a matrix, and scaling a matrix. This can be achieved by $\mathcal{O}(m\sqrt{n})$ time on an $n \times n$ matrix with m nonzeros [97]. There are different algorithms for scaling; see the survey by Idel [65], and more recent papers [3, 21, 25]. The most recent methods based on convex optimization techniques [3, 25] have the smallest run time complexity $\tilde{\mathcal{O}}(mn + n^{7/3})$ and $\tilde{\mathcal{O}}(m^3/2)$—where the terms involving the deviation from the required row/column sums and $\log n$ are discarded. The Sinkhorn–Knopp algorithm, which is the easiest to implement, has been shown to have $\mathcal{O}(\frac{n^2 \ln n}{\varepsilon^2})$ iterations, each iteration costing $\mathcal{O}(m)$ time where ε is the allowable deviation from one. Other easy to implement variants are given elsewhere [79, 80]. To the best of our knowledge, these algorithms are not yet shown to operate in fully polynomial time–there are results using the second singular value of the final matrix; and there is no run time analysis with respect to ε. As verified by our experiments in the previous chapters, Sinkhorn–Knopp kind of algorithms work well for scaling 0-1 matrices for practical purposes; usually, a few iterations suffice to obtain practically well behaving algorithms. In summary, the worst case time complexity of Algorithm 5.1 is $\tilde{\mathcal{O}}(n(\sqrt{n}m + mn^2 \ln n))$ when the Sinkhorn–Knopp algorithm is used for scaling.

The algorithm identifies a perfect matching at the end. Since the scaling factors depend on the identified perfect matching, we use $\mathbf{R}^{(\sigma,i)}$ and $\mathbf{C}^{(\sigma,i)}$ to denote the scaling matrices at the ith step of the algorithm, where the random perfect matching σ is returned. Recall that $\sigma_1^{(i)}$ denotes the column chosen at the ith step. Note that the size of the scaling matrices reduces by

Algorithm 5.1: ESTSCALINGPERM: Permanent estimation

Input: an $n \times n$, 0-1 matrix $\mathbf{A}$.

Output: Permanent estimate $X_{\mathbf{A}}$; σ a perfect matching, where the ith entry shows the column chosen for the ith row.

1: $X_{\mathbf{A}} \leftarrow 1$
2: $\mathbf{A}^{(1)} \leftarrow \mathbf{A}$
3: **for** $i = 1$ **to** n **do**
4: Filter out those entries of $\mathbf{A}^{(i)}$ that cannot be put into a perfect matching
5: $[\mathbf{R}^{(\sigma,i)}, \mathbf{C}^{(\sigma,i)}] \leftarrow \textsc{Scale}(\mathbf{A}^{(i)})$
6: Pick a random nonzero column $j \in \mathbf{A}^{(i)}(1,:)$ by using the probability density function

$$p_j = \frac{s_{1,j}}{\Sigma_{k \in \mathbf{A}^{(i)}(1,:)} s_{1,k}} \text{for all nonzeros } a_{1,j}^{(i)}$$

where $s_{t,k} = r_t^{(\sigma,i)} \cdot c_k^{(\sigma,i)}$ is the corresponding entry in the scaled matrix $\mathbf{S} = \mathbf{R}^{(\sigma,i)}\mathbf{A}^{(i)}\mathbf{C}^{(\sigma,i)}$

7: $X_{\mathbf{A}} \leftarrow X_{\mathbf{A}}/p_j$
8: $\mathbf{A}^{(i+1)} \leftarrow \mathbf{A}_{1j}^{(i)}$ {delete the first row and the jth column of $\mathbf{A}^{(i)}$}
9: $\sigma(i) \leftarrow j$ {assuming the original numbering}

one at each step, and that the first entry $r_1^{(\sigma,i)}$ of $\mathbf{R}^{(\sigma,i)}$ is the scaling factor associated with the first row of $\mathbf{A}^{(i)}$. With this notation, we can write

$$X_{\mathbf{A}} = \frac{1}{\prod_{i=1}^{n} r_1^{(\sigma,i)} \cdot c_{\sigma_1^{(i)}}^{(\sigma,i)}} \,. \tag{5.2}$$

5.3.2 The analysis

Here we give theoretical properties of the proposed algorithm. We start by showing that it obtains an unbiased estimator.

Theorem 5.1. *Let $X_{\mathbf{A}}$ be a random variable returned by Algorithm 5.1 for the estimate of the permanent of an $n \times n$, 0–1 matrix $\mathbf{A}$. Then $\mathbb{E}[X_{\mathbf{A}}] = Per(\mathbf{A})$.*

Proof. We prove the theorem using induction. For the base case where $n = 1$, the argument trivially holds. As the inductive hypothesis, assume that the argument holds for $(n-1) \times (n-1)$ matrices. Now let us consider an $n \times n$ matrix $\mathbf{A}$.

$$\begin{aligned}
\mathbb{E}[X_{\mathbf{A}}] &= \sum_{j: a_{1,j} \neq 0} p_j \cdot \frac{1}{p_j} \cdot \mathbb{E}[X_{\mathbf{A}_{1j}}] \\
&= \sum_{j: a_{1,j} \neq 0} \mathbb{E}[X_{\mathbf{A}_{1j}}] \\
&= \sum_{j: a_{1,j} \neq 0} Per(X_{\mathbf{A}_{1j}}) \text{ by the inductive hypothesis} \\
&= Per(\mathbf{A}) \,.
\end{aligned}$$

□

Next, we focus our attention on the analysis of the $\mathbb{E}[X_{\mathbf{A}}^2]$ value. For this purpose, we first state and prove the following lemma.

Lemma 5.2. *Let* $\mathbf{A}$ *be* $n \times n$ *matrix such that* $a_{1,j} = 1$, $\mathbf{A}_{1j}$ *be the* $(n-1) \times (n-1)$ *principal submatrix, and* $\mathbf{B}$ *be the* $(n-1) \times (n-1)$ *submatrix obtained from* $\mathbf{A}_{1j}$ *after discarding entries that cannot be put in a perfect matching. Let* $\mathbf{R}$, $\mathbf{C}$, $\mathbf{D}$, *and* $\mathbf{F}$ *be the positive diagonal matrices such that* $\mathbf{RAC}$ *and* $\mathbf{DBF}$ *are doubly stochastic. Then,*

$$\prod_{i=2}^{n} r_i \prod_{i=1, i\neq j}^{n} c_i \leq e^{r_1 \cdot c_j - 1} \prod_{i=1}^{n-1} d_i \cdot f_i \ .$$

Proof. Let $\mathbf{r}'$ and $\mathbf{c}'$ be the vectors $\mathbf{r}' = [r_2, \ldots, r_n]^T$ and $\mathbf{c}' = [c_1, \ldots, c_{j-1}, c_{j+1}, \ldots, c_n]^T$. Observe that for Function (5.1),

$$g_{\mathbf{B}}(\mathbf{d}, \mathbf{f}) \leq g_{\mathbf{B}}(\mathbf{r}', \mathbf{c}')$$

should hold, since $\mathbf{D}$ and $\mathbf{F}$ scale $\mathbf{B}$ to a doubly stochastic form. Therefore,

$$\mathbf{d}^T\mathbf{Bf} - \sum_{i=1}^{n-1} \ln d_i - \sum_{i=1}^{n-1} \ln f_i \leq \mathbf{r}'^T\mathbf{Bc}' - \sum_{i=1}^{n-1} \ln r'_i - \sum_{i=1}^{n-1} \ln c'_i \ ,$$

which we arrange as

$$\mathbf{d}^T\mathbf{Bf} - \mathbf{r}'^T\mathbf{Bc}' \leq \sum_{i=1}^{n-1} \ln d_i + \sum_{i=1}^{n-1} \ln f_i - \sum_{i=1}^{n-1} \ln r'_i - \sum_{i=1}^{n-1} \ln c'_i. \tag{5.3}$$

The left hand side can be bounded below. Since $\mathbf{D}$ and $\mathbf{F}$ scale $\mathbf{B}$ to a doubly stochastic form,

$$\mathbf{d}^T\mathbf{Bf} = n - 1 \ . \tag{5.4}$$

Since $\mathbf{B}$ is obtained by discarding some (positive) entries in $\mathbf{A}_{1j}$ we have

$$\mathbf{r}'^T\mathbf{Bc}' \leq \mathbf{r}'^T\mathbf{A}_{1j}\mathbf{c}' = n - 2 + r_1 \cdot c_j. \tag{5.5}$$

Hence, $\mathbf{d}^T\mathbf{Bf} - \mathbf{r}'^T\mathbf{Bc}' \geq 1 - r_1 \cdot c_j$. Note that this last inequality will be tight, if we do not get rid of any entries from $\mathbf{A}_{1j}$, in which case $\mathbf{B} = \mathbf{A}_{1j}$. Furthermore, since $0 < r_1 \cdot c_j \leq 1$, we see that

$$0 \leq 1 - r_1 \cdot c_j \leq \mathbf{d}^T\mathbf{Bf} - \mathbf{r}'^T\mathbf{Bc}'.$$

By combining this with (5.3) and exponentiation of all parts, we obtain

$$1 \leq e^{1 - r_1 \cdot c_j} \leq e^{\mathbf{d}^T\mathbf{Bf} - \mathbf{r}'^T\mathbf{Bc}'} \leq \frac{\prod_{i=1}^{n-1} d_i \cdot f_i}{\prod_{i=1}^{n-1} r'_i \cdot c'_i}, \tag{5.6}$$

and this concludes the proof. □

Lemma 5.2 is important as it relates the scaling coefficients of the matrices between successive steps of Algorithm 5.1. The selection at the ith step can lead to many possible different matrices, all of which must be taken into consideration for the analysis of $\mathbb{E}[X_{\mathbf{A}}^2]$. The lemma provides

a common bound for all of them, which is used to simplify the analysis. We can now proceed with the proof of our main theorem.

Theorem 5.2. *Let* $\mathbf{A}$ *be* $n \times n$ *matrix with total support, and* $\mathbf{RAC}$ *be its doubly stochastic scaling. Then* $\mathbb{E}[X_{\mathbf{A}}^2] \leq \frac{1}{\prod_i r_i \cdot c_i} \cdot Per(\mathbf{A})$.

Proof. We prove the theorem by induction. The base case $n = 1$ holds trivially. Assume that the theorem holds for $(n-1) \times (n-1)$ matrices. We then have

$$\mathbb{E}[X_{\mathbf{A}}^2] = \sum_{a_{1,j} \neq 0} r_1 \cdot c_j \cdot \left(\frac{1}{r_1^2 \cdot c_j^2} \cdot \mathbb{E}[X_{\mathbf{A}_{1j}}^2] \right)$$

$$\mathbb{E}[X_{\mathbf{A}}^2] = \sum_{a_{1,j} \neq 0} \frac{1}{r_1 \cdot c_j} \cdot \mathbb{E}[X_{\mathbf{A}_{1j}}^2]$$

$$\mathbb{E}[X_{\mathbf{A}}^2] \leq \sum_{a_{1,j} \neq 0} \frac{1}{r_1 \cdot c_j} \cdot \frac{1}{\prod_z d_z \cdot f_z} \cdot Per(\mathbf{A}_{1j})$$

by the inductive hypothesis, where $\mathbf{D}$ and $\mathbf{F}$ scale $\mathbf{A}_{1j}$

$$\mathbb{E}[X_{\mathbf{A}}^2] \leq \sum_{a_{1,j} \neq 0} \frac{1}{r_1 \cdot c_j} \cdot \frac{1}{\prod_{z=2}^n r_z \cdot \prod_{z=1, z \neq j}^n c_z} \cdot Per(\mathbf{A}_{1j}) \quad \text{by Lemma 5.2,}$$

$$\mathbb{E}[X_{\mathbf{A}}^2] \leq \frac{1}{\prod_i r_i \cdot c_i} \cdot \sum_{a_{1,j} \neq 0} Per(\mathbf{A}_{1j})$$

$$\mathbb{E}[X_{\mathbf{A}}^2] \leq \frac{1}{\prod_i r_i \cdot c_i} \cdot Per(\mathbf{A}) \,.$$

□

In the above proof, we made use of a weakened version for Lemma 5.2. In the original lemma, $\frac{\prod_i d_i \cdot f_i}{\prod_{i \neq 1} r_i \prod_{i \neq j} c_i} \leq e^{r_1 \cdot c_j - 1} \leq 1$ for the submatrix $\mathbf{B} = \mathbf{A}_{1j}$ with scaling coefficients $\mathbf{d}, \mathbf{f}$. Because each possible j has a different exponent term associated with it, we had to replace all exponent terms by the common upper bound of one, so as to obtain a bound over all perfect matchings. These exponential terms can often be significantly less than one, and this replacement can lead to a loose upper bound for $\mathbb{E}[X_{\mathbf{A}}^2]$. We proceed to state two theorems in which the exponent term is handled differently in order to obtain more accurate bounds.

Theorem 5.3. $\mathbb{E}[X_{\mathbf{A}}^2] \leq Per(\mathbf{A}) \cdot mean \left(\sum_\sigma e^{\sum_j r_1^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)}} \right) \cdot \frac{e^{-n}}{\prod_i r_i \cdot c_i}$ *where the sum is over all perfect matchings* σ.

Proof. Consider any perfect matching σ of $\mathbf{A}$. Let $X_{\mathbf{A}_\sigma} = \frac{1}{\prod_{i=1}^n r_1^{(\sigma,i)} c_{\sigma_1^{(i)}}^{(\sigma,i)}}$, which is equivalent to the value of the random variable $X_{\mathbf{A}}$ should the matching σ be returned, as shown in (5.2).

We use a bottom-up induction to prove the following: Let $1 \leq i \leq n$. Then

$$\prod_{j=i}^n \frac{1}{r_1^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)}} \leq \frac{e^{\left(\sum_{j=i}^n r_1^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)} \right) - n - 1 + i}}{\prod_{j=i}^n r_{j-i+1}^{(\sigma,i)} c_{\sigma_1^{(j)}}^{(\sigma,i)}} \,. \tag{5.7}$$

That is we provide a relation for the scaling matrices $\mathbf{R}^{(\sigma,i)}$ and $\mathbf{C}^{(\sigma,i)}$ of the ith step and the multiplication of all $r_1^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)}$ where $j \geq i$. We use the index $j - i + 1$ to refer to rows with index greater than $i \geq 1$ because in the reshaped matrix of the ith step, row i occupies the first position. The idea resembles the proof of Theorem 5.2 except that here a tighter upper bound is provided by having the numerator less than one.

In the base case $i = n$, and the relation holds with equality. Assume it holds until i where $i > 1$. Then, for $i - 1$

$$\prod_{j=i-1}^{n} \frac{1}{r_1^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)}} \leq \frac{e^{\left(\sum_{j=i}^{n} r_1{}^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)}\right) - n - 1 + i}}{\prod_{j=i}^{n} r_{j-i+1}^{(\sigma,i)} c_{\sigma_1^{(j)}}^{(\sigma,i)}} \cdot \frac{1}{r_1^{(\sigma,i-1)} c_{\sigma_1^{(i-1)}}^{(\sigma,i-1)}}$$

by the induction hypothesis holding for value i.

Note that $\prod_{j=i}^{n} r_{j-i+1}^{(\sigma,i)} c_{\sigma_1^{(j)}}^{(\sigma,i)}$ is the product of the scaling factors at the ith step, and that $\prod_{j=i}^{n} r_{j-(i-1)+1}^{(\sigma,i-1)} c_{\sigma_1^{(j)}}^{(\sigma,i-1)}$ is the product of the scaling factors at the $(i-1)$st step excluding the row scaling entry $r_1^{(\sigma,i-1)}$ and the associated column scaling entry. Therefore, we can replace

$$\frac{1}{\prod_{j=i}^{n} r_{j-i+1}^{(\sigma,i)} c_{\sigma_1^{(j)}}^{(\sigma,i)}} \quad \text{with} \quad \frac{e^{\left(r_1{}^{(\sigma,i-1)} c_{\sigma_1^{(i-1)}}^{(\sigma,i-1)}\right) - 1}}{\prod_{j=i}^{n} r_{j-(i-1)+1}^{(\sigma,i-1)} c_{\sigma_1^{(j)}}^{(\sigma,i-1)}}$$

using Lemma 5.2 to obtain an upper bound. That is

$$\prod_{j=i-1}^{n} \frac{1}{r_1^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)}} \leq \frac{e^{\sum_{j=i}^{n} r_1{}^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)} - n - 1 + i} \cdot e^{r_1{}^{(\sigma,i-1)} c_{\sigma_1^{(i-1)}}^{(\sigma,i-1)} - 1}}{\prod_{j=i}^{n} r_{j-i+2}^{(\sigma,i-1)} c_{\sigma_1^{(j)}}^{(\sigma,i-1)}} \cdot \frac{1}{r_1^{(\sigma,i-1)} c_{\sigma_1^{(i-1)}}^{(\sigma,i-1)}} .$$

We see that in both terms of the fraction we can include $r_1^{(\sigma,i-1)} c_{\sigma_1^{(i-1)}}^{(\sigma,i-1)}$ to its respective aggregator

$$\prod_{j=i-1}^{n} \frac{1}{r_1^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)}} \leq \frac{e^{\sum_{j=i-1}^{n} r_1{}^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)} - n - 1 + (i-1)}}{\prod_{j=i-1}^{n} r_{j-i+2}^{(\sigma,i-1)} c_{\sigma_1^{(j)}}^{(\sigma,i-1)}}$$

which concludes the proof of (5.7), as for $j = i - 1$, we have $j - i + 2 = 1$. Thus for $i = 1$ we get:

$$X^2_{\mathbf{A}_\sigma} \leq \frac{e^{\sum_j r_1{}^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)}}}{\prod_i r_i \cdot c_i} \cdot e^{-n}.$$

Since

$$\mathbb{E}[X_{\mathbf{A}}^2] = \sum_{\sigma} X_{\mathbf{A}_\sigma}^2 \cdot \prod_{i=1}^{n} r_1^{(\sigma,i)} c_{\sigma_1^{(i)}}^{(\sigma,i)}, \quad \text{we get that}$$

$$\mathbb{E}[X_{\mathbf{A}}^2] = \sum_{\sigma} X_{\mathbf{A}_\sigma}$$

$$\mathbb{E}[X_{\mathbf{A}}^2] \leq \sum_{\sigma} \frac{e^{\sum_j {r_1}^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)}}}{\prod_i r_i \cdot c_i} \cdot e^{-n} ,$$

as we have $Per(\mathbf{A})$ permutations each with each own value. Replacing with the mean multiplied by $Per(\mathbf{A})$ concludes the theorem. □

Theorem 5.4. *Let* $\mathbf{A}$ *be* $n \times n$ *0-1 matrix with total support, and* $\mathbf{RAC}$ *be its doubly stochastic scaling. There exists a positive vector* $\mathbf{k}$ *such that* $\mathbb{E}[X_{\mathbf{A}}^2] \leq \frac{e^{v-n}}{\prod_i r_i \cdot c_i} \cdot Per(\mathbf{A})$*, where* $v = \sum_i k_i = \|\mathbf{k}\|_1 \leq n$*.*

Proof. The proof follows that of Theorem 5.2 to build $\mathbf{k}$ by considering the maximum exponent value at each step and can be found in the corresponding paper [J2]. □

Note that Theorem 5.4 implies Theorem 5.2 if we set $k_i = 1$. The theorem is constructive but does not yield a polynomial algorithm. Algorithm 5.1 can be made to construct a $\mathbf{k}$ associated with a perfect matching σ obtained at the end, to show how much $\frac{1}{\prod_{i=1}^{n} r_1^{(\sigma,i)} \cdot c_{\sigma_1^{(i)}}^{(\sigma,i)}}$ deviates from a potential bound that can be obtained by *mean* $\left(\sum_\sigma e^{\sum_j {r_1}^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)}}\right) \cdot \frac{e^{-n}}{\prod_i r_i \cdot c_i}$ from the previous theorem. Note also that $\sum_j {r_1}^{(\sigma,j)} c_{\sigma_1^{(j)}}^{(\sigma,j)}$ is less than n, unless all terms are one, in which case $\mathbf{A}$ is the $n \times n$ identity matrix (and the permanent of value one is obtained exactly). Therefore, the obtained bound shows that the upper bound stated in Theorem 5.2 is loose by exponents of e.

5.4 An estimator for undirected graphs

In this section, we investigate how to extend the proposed scaling-based approach for undirected graphs.

5.4.1 The algorithm and its analysis

The variant for undirected graphs shares similar properties with the algorithm for the bipartite case. In particular with X_G being the estimate by the algorithm and $M(G)$ being the number of perfect matchings in G, it can be shown that

$$\mathbb{E}[X_G] = M(G)$$

and

$$\mathbb{E}[X_G^2] \leq M(G)\frac{1}{\prod_i r_i} ,$$

where r_i is the ith diagonal entry of the scaling factor $\mathbf{R}$ of the adjacency matrix of G. Note that since the adjacency matrix $\mathbf{A}$ is symmetric, its scaling is in the form $\mathbf{S} = \mathbf{RAR}$, i.e., we do not need separate scaling values for rows and columns. Algorithm 5.2 shows the pseudocode of the proposed approach.

Algorithm 5.2: ESTSCALINGMTC: Estimation of the number of perfect matchings in graphs

Input: $G = (V, E)$ an undirected graph with $n = |V|$ vertices having a perfect matching.
Output: An estimate X_G of the number of perfect matchings in G.

1: $X_G \leftarrow 1$
2: $G^{(1)} \leftarrow G$
3: **for** $i = 1$ **to** n by increments of two **do**
4: Let $\mathbf{A}^{(i)}$ be the adjacency matrix of $G^{(i)}$.
5: Filter out those entries of $\mathbf{A}^{(i)}$ that cannot be put into a perfect matching in the bipartite graph corresponding to $\mathbf{A}^{(i)}$, and let $G^{(i)}$ correspond to the graph of $\mathbf{A}^{(i)}$
6: $[\mathbf{R}^{(i)}] \leftarrow$ SYMSCALE($\mathbf{A}^{(i)}$)
7: Let $\mathcal{T} \leftarrow \{j : a_{1,j}^{(i)} \neq 0 \text{ and } G^{(i)} - \{v_1, v_j\} \text{ has a perfect matching}\}$
8: Pick a random nonzero column j from $\mathcal{T}$ by using the probability density function

$$p_k = \frac{s_{1,k}}{\Sigma_{t\in\mathcal{T}} s_{1,t}}, \text{for all nonzero } a_{1,k}^{(i)} \text{ with } k \in \mathcal{T}$$

where $s_{t,k} = r_t^{(i)} \cdot r_k^{(i)}$ is the corresponding entry in the scaled matrix $\mathbf{S} = \mathbf{R}^{(i)}\mathbf{A}^{(i)}\mathbf{R}^{(i)}$
9: $X_G \leftarrow X_G/p_j$
10: $G^{(i+1)} \leftarrow G^{(i)} - \{v_1, v_j\}$ {delete the vertices v_1 and v_j from $G^{(i)}$ to obtain $G^{(i+1)}$}

At the beginning of the iteration i, we have a graph $G^{(i)}$ having at least one perfect matching. We then get the adjacency matrix $\mathbf{A}^{(i)}$ of this graph, which is symmetric. We then check if all entries in $\mathbf{A}^{(i)}$ can put into a nonzero diagonal; that is we look at the Dulmage–Mendelsohn decomposition of the bipartite graph associated with $\mathbf{A}^{(i)}$, and discard edges that cannot be in a perfect matching. We discard those entries (symmetrically) and obtain a sparser matrix $\mathbf{A}^{(i)}$ and scale the resulting matrix. This also translates to an updated $G^{(i)}$. Then, for each edge incident on the first vertex of $G^{(i)}$, at first we test if there is a perfect matching in $G^{(i)}$ containing that edge. We discard the edges not inside any perfect matching to avoid returning a zero estimate. Among all the edges which are left (i.e., that are inside at least one perfect matching), we choose an edge from a probability distribution where the probabilities are proportional to the scaling entries of the allowed edges. Notice that

$$p_k = \frac{s_{1,k}}{\Sigma_{t\in\mathcal{T}} s_{1,t}}$$

where $\mathcal{T} = \{j : a_{1,j}^{(i)} \neq 0 \text{ and } G^{(i)} - \{v_1, v_j\} \text{ has a perfect matching}\}$, $s_{1,t} = r_1^{(i)} \cdot r_t^{(i)}$, and $0 < \Sigma_{t\in\mathcal{T}} s_{1,t} \leq 1$. Therefore, $p_k \geq r_1^{(i)} \cdot r_k^{(i)}$ at Line 8 of Algorithm 5.2.

As in the case of the estimator for bipartite graphs, we begin the analysis of Algorithm 5.2 by

showing that it is an unbiased estimator. In order to do so, we adapt the proof of Theorem 5.1 appropriately for undirected graphs.

Theorem 5.5. *Let X_G be a random variable returned by Algorithm 5.2. Then $\mathbb{E}[X_G] = M(G)$, where $M(G)$ represents the number of perfect matchings in the graph G.*

Proof. We prove the claim via induction. As the base-case, we consider $n = 2$ and the argument holds where the adjacency matrix is $\mathbf{A} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. Assume that the inductive hypothesis holds for $n-2$. Let G_{ij} be obtained by removing vertices v_i and v_j from G, which corresponds to deleting the rows and the columns i, j of the adjacency matrix of G to form the $(n-2) \times (n-2)$ adjacency matrix $\mathbf{A}_{ij}$ of G_{ij}. We have the following

$$\begin{aligned}
\mathbb{E}[X_G] &= \sum_{j:a_{1,j}\neq 0} p_j \cdot \frac{1}{p_j} \cdot \mathbb{E}[X_{G_{ij}}] \\
&= \sum_{j:a_{1,j}\neq 0} \mathbb{E}[X_{G_{ij}}] \\
&= \sum_{j:a_{1,j}\neq 0} M(G_{ij}) \text{ by the inductive hypothesis} \\
&= M(G) \ .
\end{aligned}$$

□

Having shown that X_G is an unbiased estimator, we now proceed to bound $\mathbb{E}[X_G^2]$ and obtain a bound on the number of samples required for an (ε, δ)-approximation. For this, we again make use of a lemma that associates the scaling coefficients at different steps of Algorithm 5.2 with each other.

Lemma 5.3. *Let $\mathbf{A}$ be a symmetric $n \times n$ doubly stochastically scalable matrix with $\alpha_{1j} = \alpha_{j1} = 1$ with all diagonal values zero. Let $\mathbf{R}$ be its scaling matrix. Assume we remove the rows and columns with indices 1 and j from $\mathbf{A}$. We then discard the entries that are not in support to obtain $\mathbf{B}$, which is a symmetric scalable matrix of size $n-2$. Let $\mathbf{D}$ be $\mathbf{B}$'s scaling matrix, i.e., $\mathbf{DBD}$ is doubly stochastic. Then*

$$\prod_{k=1,k\neq 1,j}^{n} r_k \leq e^{r_1 r_j - 1} \prod_{z=1}^{n-2} d_z \ .$$

Proof. The proof is similar to the proof of Lemma 5.2 and can be found in the corresponding paper [J2]. □

Theorem 5.6 now shows the equivalent of Theorem 5.2 for undirected graphs and completes the analysis of Algorithm 5.2.

Theorem 5.6. *Let G be an undirected graph, $\mathbf{A}$ be the $n \times n$ adjacency matrix of G with all supported nonzeros, and $\mathbf{R}$ be the diagonal matrix which scales $\mathbf{A}$ into the doubly stochastic form, e.g., $\mathbf{RAR}$ is doubly stochastic. Then*

$$\mathbb{E}[X_G^2] \leq M(G) \cdot \frac{1}{\prod_i r_i} \ .$$

Proof. We prove the theorem by induction. The base case $n = 2$ holds trivially. Let the inductive hypothesis be that the argument holds for graphs with $n - 2$ vertices.

$$\mathbb{E}[X_G^2] = \sum_{a_{i,j} \neq 0} p_j \cdot \left(\frac{1}{p_j^2} \cdot \mathbb{E}[X_{G_{ij}}^2] \right)$$

$$\mathbb{E}[X_G^2] = \sum_{a_{i,j} \neq 0} \frac{1}{p_j} \cdot \mathbb{E}[X_{G_{ij}}^2]$$

$$\mathbb{E}[X_G^2] \leq \sum_{a_{i,j} \neq 0} \frac{1}{r_i \cdot r_j} \cdot \mathbb{E}[X_{G_{ij}}^2]$$

because $p_j \geq r_i \cdot r_j$ by the definition p_j at Line 8 of Algorithm 5.2,

$$\mathbb{E}[X_G^2] \leq \sum_{a_{i,j} \neq 0} \frac{1}{r_i \cdot r_j} \cdot \frac{1}{\prod_z d_z} \cdot M(G_{ij})$$

by the inductive hypothesis, where $\mathbf{D}\mathbf{A}_{(ij)}\mathbf{D}$ is doubly stochastic, where $\mathbf{A}_{(ij)}$ is the filtered adjacency matrix of G_{ij}. Hence,

$$\mathbb{E}[X_G^2] \leq \sum_{a_{i,j} \neq 0} \frac{1}{r_i \cdot r_j} \cdot \frac{1}{\prod_{z \neq i, z \neq j}^{n} r_z} \cdot M(G_{ij}) \quad \text{by Lemma 5.3,}$$

$$\mathbb{E}[X_G^2] \leq \frac{1}{\prod_z r_z} \cdot \sum_{a_{i,j} \neq 0} M(G_{ij})$$

$$\mathbb{E}[X_G^2] \leq \frac{1}{\prod_z r_z} \cdot M(G) \ .$$

□

5.4.2 Filtering out redundant edges

A complication in the undirected case is that to eliminate edges that do not belong to any perfect matching it is not sufficient to only use the Dulmage–Mendelsohn decomposition of $\mathbf{A}$. We demonstrate this with the following fact.

Fact 5.1. *Let G be an undirected graph, and $\mathbf{A}$ be the $n \times n$ adjacency matrix of G. Then $Per(\mathbf{A})$ is larger than or equal to $M(G)$.*

Proof. The fact holds trivially in the case that $M(G) = 0$. We hence assume in the following that $M(G) > 0$. Let G' be the bipartite graph with $2n$ vertices obtained from $\mathbf{A}$. Then for each perfect matching in $G = (V, E)$, there is a corresponding perfect matching in $G' = (V', E')$. Let $\{(u_1, u_2), \ldots, (u_{n-1}, u_n)\}$ be a perfect matching in G where $u_i \in V$ for $i \in \{1, \ldots, n\}$. Let the vertices in the first and the second part of the bipartite graph G' be denoted as v_is and w_is, respectively, where $v_i \in V'$ and $w_i \in V'$ for $i \in \{1, \ldots, n\}$.

Since the edge (u_i, u_{i+1}) is in the matching for $i \in \{1, \ldots, n\}$, it is in G. Therefore $a_{i,i+1} = a_{i+1,i} = 1$ are nonzeros in $\mathbf{A}$. Hence, in the bipartite graph G', we have the edges $(v_i, w_{i+1}) \in E'$ and $(v_{i+1}, w_i) \in E'$. From these edges, a perfect matching can be constructed in G'.

Thus for each matching in G, one can construct a perfect matching in G' and the number of the perfect matchings in G' is equal to $Per(\mathbf{A})$. Hence, $Per(\mathbf{A})$ is at least as large as $M(G)$. □

The above fact shows that any off-diagonal edge in the Dulmage–Mendelsohn decomposition of $\mathbf{A}$ cannot belong to a perfect matching in G, but it does not help us to eliminate possible edges that do not belong to such symmetrical matchings (v_i, w_j) and (w_j, v_i). These edges may be cleared by using an exact algorithm for computing maximum matchings to detect whether selecting one of them leads to a perfect matching for the remaining vertices or not. One may ask if it is possible to use the Gallai–Edmonds [86] decomposition, which partially extends the Dulmage–Mendelsohn decomposition to undirected graphs. This decomposition can be used to find perfectly matchable sub-graphs and odd components if there is no perfect matching in the graph. It does not give any useful information for our purposes in graphs with perfect matchings, while the Dulmage–Mendelsohn decomposition for bipartite graphs states which edges cannot be put into a perfect matching.

5.5 Experiments

The experiments are performed on a machine equipped with an Intel Core i7-7600 CPU and with 16 GB of available ram. For the implementation, we used MATLAB 2017. In the next two subsections, we present the experimental results on bipartite graphs and general, undirected graphs, respectively.

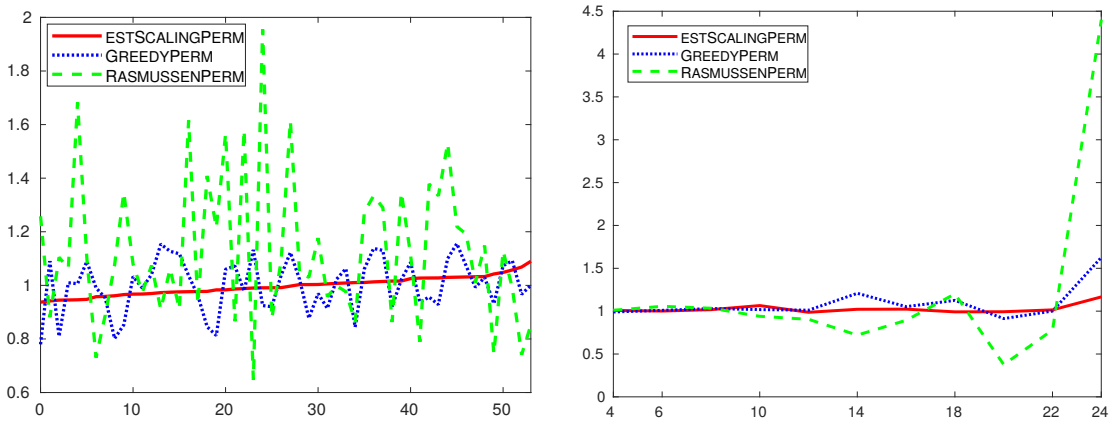
5.5.1 On bipartite graphs

To see how the proposed algorithm estScalingPerm fares in practice on bipartite graphs, we compare it against the original estimator of Rasmussen as well as the approach due to Fürer and Kasiviswanathan [49]. We shall refer to latter's variant for bipartite graphs as GreedyPerm. We used an improved version of Rasmussen's randomized algorithm in which edges that do not participate in perfect matchings are discarded. We shall refer to it as RasmussenPerm. Additionally, all three algorithms process the rows in increasing order of nonzeros, adjusted at each step, to select a random entry. For each test, we take 1000 samples and report their mean.

5.5.1.1 Random and synthetic datasets

For the first set of bipartite graph experiments, we consider random matrices of size 40 and sparsity $\frac{4}{n}$, in other words there are about 160 nonzeros. We used an exact (exponential time) algorithm [95] to compute the permanents. These matrices can have large permanents (e.g., around 10^7) and are among the largest an exact non-parallel algorithm, which simply enumerates all the perfect matchings, can handle. The results are summarized in Figure 5.1(a). In this figure, we display the ratio of the estimate to the exact value. As seen in the figure, our approach almost always has good performance. In contrast, the RasmussenPerm estimator often obtains results that are worse than the other two approaches (even if modified to avoid returning zeros). The GreedyPerm estimator exhibits a better performance than RasmussenPerm, while estScalingPerm is better than GreedyPerm (it has a smaller and less frequent deviation from the value of the permanent).

To provide results with larger n, we focus on the class of matrices which correspond to grids as the second set experiments on bipartite graphs. For these matrices, an exact formula for the permanent is given independently by Kasteleyn [74] and Temperley and Fisher [103]. The



(a) Sparse 40×40 matrices.

(b) Grid graphs.

Figure 5.1 – Comparison of the approximation ratios of ESTSCALINGPERM, RASMUSSENPERM and GREEDYPERM (a) on 54 random sparse graphs with $n = 40$ and sparsity factor $4/n$ in increasing order of the approximation ratio of the proposed estimator; and (b) on square grids of even length in increasing order of side length.

number of perfect matchings in an $m \times n$ grid is given by the formula

$$\prod_{j=1}^{m} \prod_{k=1}^{n} \left(4\cos^2\left(\frac{\pi j}{m+1}\right) + 4\cos^2\left(\frac{\pi k}{n+1}\right)\right)^{1/4} .$$

The results presented in Figure 5.1(b) concur with the previous test. We observe that ESTSCALINGPERM seems to provide better performance than the two other alternatives. This is particularly notable in the last case of the 24×24 grid where only ESTSCALINGPERM manages to obtain an estimate in close range of the actual answer. This suggests that the assignment of probabilities via a doubly stochastic scaling method makes the overall procedure more reliable.

As the third set of experiments bipartite graphs, we give the results for 1000 simulations of the three approaches on the matrix of the 36×36 grid to examine in detail how the algorithms behave for larger graphs. The results are presented in Figure 5.2. To draw this figure, we used MATLAB's `histfit` command on the 1000 estimates of each algorithm, to plot a histogram and fit a bell curve so as to understand the distribution. As the values are very large, we present the results on a log-scale. In this figure, we observe less variation between the independent runs of the proposed method. Furthermore, the approximation factor of the mean estimate of ESTSCALINGPERM, 1.11, is significantly better than those of the other two approaches; GREEDYPERM's approximation ratio is 0.42 while RASMUSSENPERM's is worse than both obtaining a 0.0028 approximation (the approximation ratios are obtained using the exact formula given above).

5.5.1.2 Real-life bipartite graphs

To further evaluate the practical performance of the proposed approximation scheme, we used a set of matrices from the SuiteSparse Matrix Collection [31]. The properties of the matrices can be seen in Table 5.1.

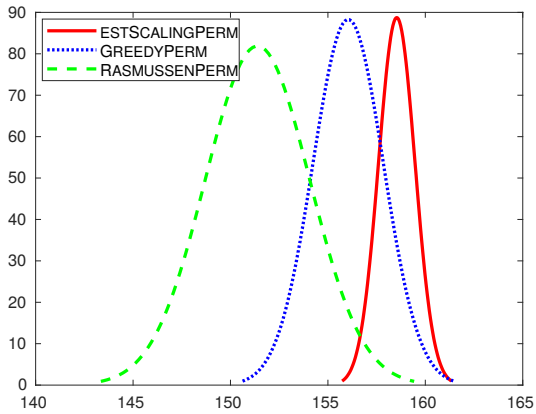


Figure 5.2 – The performance of the estimators on the 36×36 grid. The logarithms of the 1000 estimator values are presented on the x-axes and the y-axes demonstrate the distribution of the samples for the values in each approach. We note that the logarithm of the actual permanent is 159.49.

Name	n	nnz	permanent	RASMUSSENPERM	GREEDYPERM	ESTSCALINGPERM
`Grund/dss`	53	144	$2.93 \cdot 10^{5}$	1.080	0.949	1.033
`DIMACS10/chesapeake`	39	340	$1.32 \cdot 10^{13}$	0.302	0.918	0.993
`HB/bcsstk01`	48	400	$2.01 \cdot 10^{25}$	0.973	0.966	0.993
`HB/impcolb`	59	271	$1.11 \cdot 10^{08}$	1.027	0.824	0.985
`HB/will57`	57	281	$1.07 \cdot 10^{18}$	1.578	0.944	1.022
`HB/dwt59`	59	267	$4.64 \cdot 10^{18}$	1.367	0.877	0.952
`AG-Monien/netz4504dual`	615	2342	Permanent is too expensive to compute			
`Bai/dw256A`	512	2480				
`Bai/dw256B`	512	2500				
`HB/662bus`	662	2474				
`HB/685bus`	685	3249				
`JGDHomology/ch5-5-b3`	600	2400				
`VDOL/dynamicSoaringProblem1`	647	5367				

Table 5.1 – Names, dimensions, and numbers of nonzeros of the real-life square matrices corresponding to bipartite graphs used to evaluate the performance of the estimators. For the first set of six smaller matrices with $n < 60$, the exact permanent value and the approximation of the estimators with 1000 samples are also given.

With known permanents. For the first set of experiments, we used six small real-life square matrices with $n < 60$. For these matrices, it is possible to calculate the permanent exactly using a recent parallel algorithm [75]. Table 5.1 additionally exhibits the approximation found by each of the three algorithms after performing 1000 runs and taking the mean. The results are similar to those with synthetic and random matrices. estScalingPerm obtains the best performance, followed by GreedyPerm, and RasmussenPerm, which has the worst behavior overall. In more detail, estScalingPerm obtained a mean deviation of 2% from the permanent, whereas GreedyPerm had an average deviation of 9%. For `impcolb`, estScalingPerm was 16.1% closer to the permanent than GreedyPerm, which amounted to a difference of $1.7{\times}10^7$ between the two approximations. For `dwt59`, the difference between the answers of estScalingPerm and GreedyPerm was $3.48 \cdot 10^{17}$ in favor of estScalingPerm. RasmussenPerm's average deviation from the permanent at 39% was much worse than the other methods.

To demonstrate the effectiveness of estScalingPerm in more detail, Figure 5.3 presents a histogram of the estimators and fits a bell curve. We additionally show the value of the permanent with a vertical line to display how the estimators are spread around it. As it can be observed from all six subfigures, estScalingPerm's estimations are more closely concentrated around the value of the permanent, and they additionally span the smallest interval.

With unknown permanents. As the last set of bipartite graph experiments, we present the performance of the three approaches with 1000 estimations on seven larger matrices with $n > 500$ from Table 5.1. For these matrices, we do not know the exact permanent values, and we plot only the estimates in Figure 5.4. As before, the figures were generated with MATLAB's `histfit` command, and the results are given in log-scale. In all cases, we see that the values obtained by RasmussenPerm's approach span a larger interval than the other two, and GreedyPerm has a larger span than estScalingPerm. We also note that the distributions are similar to those in Figure 5.3, which is another indication that estScalingPerm most probably obtains better estimates than both GreedyPerm and RasmussenPerm on these graphs as well. We also present the mean, standard deviation (std), and the std/mean ratio (i.e., coefficient of variation) of the estimators in Table 5.2. We calculated the standard deviation using the formula $\sigma^2 = \dfrac{\sum_{i=1}^{N}(X_i - \overline{X})^2}{N-1}$ where N represents the number of samples and X_i corresponds to the ith sample with $\overline{X}$ being their mean value.

A trend we notice in all matrices is that the RasmussenPerm algorithm always obtains the smallest mean value, whereas the proposed algorithm returns the largest. Furthermore, the std/mean ratio of estScalingPerm is almost always the smallest of the three (except for the matrix `dw256B`). By combining these observations with the previous ones, we conclude that estScalingPerm's estimations are more concentrated around the returned mean compared to the other two algorithms. In other words, the proposed approach has lesser variation than the other two.

The run times of estScalingPerm on these seven matrices are very close to those of the other two approaches. The average run time to execute an estimator using estScalingPerm is 0.92 seconds, whereas RasmussenPerm and GreedyPerm require 0.13 and 0.14 seconds, respectively.

We do note however that because of the matrix scaling, for larger matrices the time differences between estScalingPerm and the other two heuristics should grow larger. That is understandable, as the run time of algorithms for matrix scaling is proportional to the number of nonzeros, whereas the other two heuristics require very little preprocessing work at each step

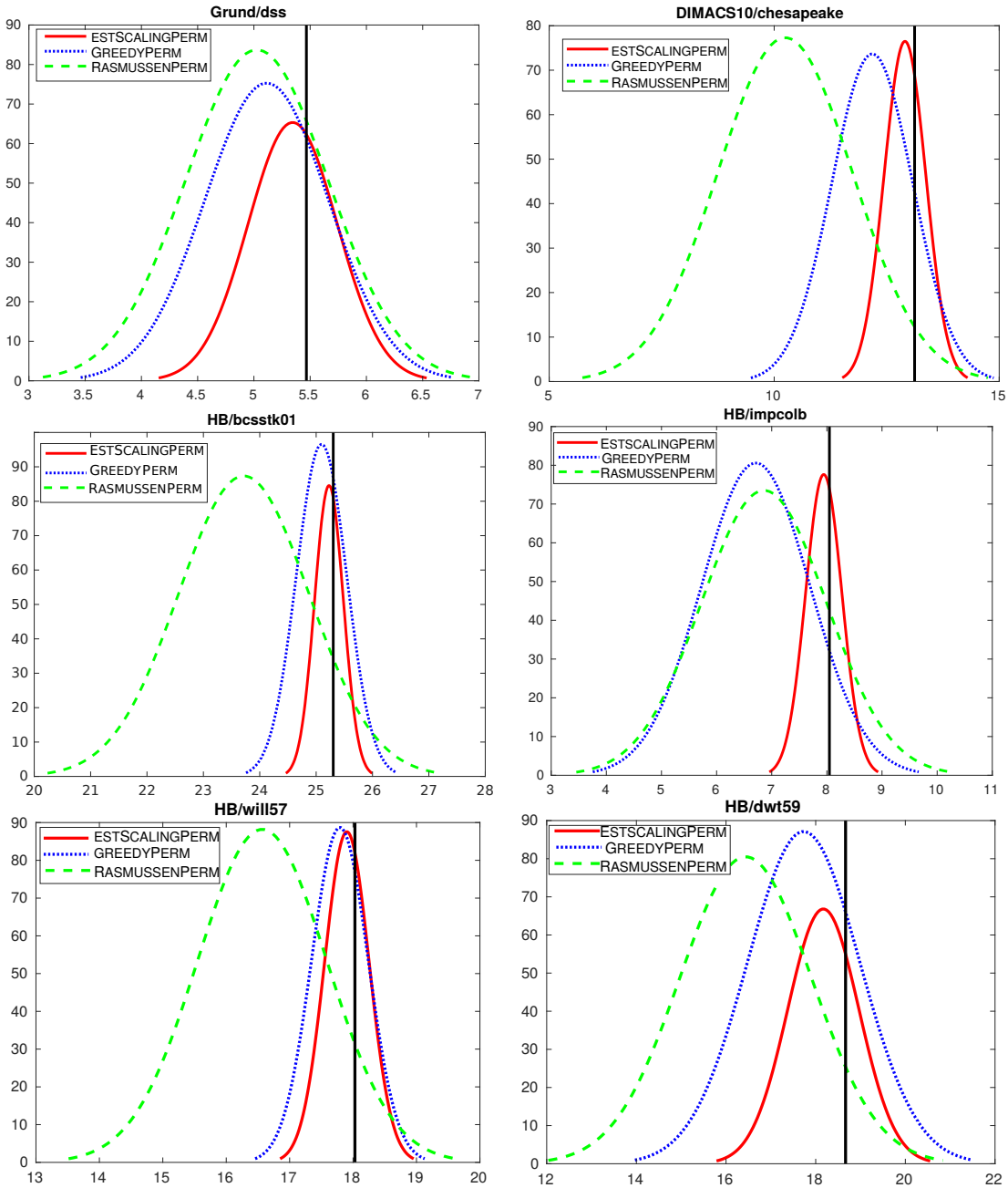


Figure 5.3 – The performance of the estimators on the six matrices with $n < 60$ given in Table 5.1. The logarithms of the 1000 estimator values are presented on the x-axes and the y-axes demonstrate the distribution of the samples for the values in each approach. The vertical line corresponds to the actual value of the permanent.

Matrix	RasmussenPerm			GreedyPerm			estScalingPerm		
	mean	std	$\frac{\text{std}}{\text{mean}}$	mean	std	$\frac{\text{std}}{\text{mean}}$	mean	std	$\frac{\text{std}}{\text{mean}}$
`netz4504dual`	$5.12 \cdot 10^{140}$	$1.60 \cdot 10^{142}$	31.36	$7.68 \cdot 10^{142}$	$1.68 \cdot 10^{144}$	21.86	$3.54 \cdot 10^{143}$	$3.01 \cdot 10^{144}$	8.52
`dw256A`	$3.59 \cdot 10^{158}$	$1.13 \cdot 10^{160}$	31.54	$2.77 \cdot 10^{162}$	$3.75 \cdot 10^{163}$	13.56	$2.11 \cdot 10^{164}$	$2.64 \cdot 10^{165}$	12.53
`dw256B`	$7.14 \cdot 10^{158}$	$2.05 \cdot 10^{160}$	28.76	$3.86 \cdot 10^{162}$	$4.49 \cdot 10^{163}$	11.63	$4.02 \cdot 10^{164}$	$5.29 \cdot 10^{165}$	13.15
`662bus`	$1.51 \cdot 10^{142}$	$4.69 \cdot 10^{143}$	31.07	$6.48 \cdot 10^{150}$	$1.45 \cdot 10^{152}$	22.32	$1.41 \cdot 10^{151}$	$2.83 \cdot 10^{152}$	20.09
`685bus`	$2.75 \cdot 10^{187}$	$7.56 \cdot 10^{188}$	27.49	$1.04 \cdot 10^{203}$	$2.86 \cdot 10^{204}$	27.55	$1.49 \cdot 10^{202}$	$2.59 \cdot 10^{203}$	17.37
`ch5-5-b3`	$1.80 \cdot 10^{136}$	$3.23 \cdot 10^{137}$	17.99	$3.92 \cdot 10^{137}$	$7.31 \cdot 10^{138}$	18.62	$5.19 \cdot 10^{138}$	$6.00 \cdot 10^{139}$	11.57
`dynamicSoar1`	$8.27 \cdot 10^{254}$	$2.46 \cdot 10^{256}$	29.80	$1.09 \cdot 10^{259}$	$2.91 \cdot 10^{260}$	26.70	$3.29 \cdot 10^{267}$	$4.38 \cdot 10^{268}$	13.32

Table 5.2 – Statistics for the seven larger bipartite graphs. The original name of the matrix corresponding to the last one is `dynamicSoaringProblem1` and is shortened to fit into the column. The preceding family names of the matrices are not shown.

aside from removing entries without support from the matrix. Consequently, for the estScalingPerm approach there is a trade-off between accuracy and run time. For graphs of similar size to those seen in Table 5.1, this trade-off does not lead to significant differences. In order to deal with larger graphs, we can reduce the number of scaling iterations at the expense of some loss in accuracy or resort to parallelism to speed-up the scaling procedure.

5.5.2 On general, undirected graphs

We examine the performance of the estimators on five undirected graphs obtained from matrices available in the SuiteSparse Matrix Collection. After downloading the matrices, we made them pattern-wise symmetric by adding all missing symmetric entries. Furthermore, we discarded the values in the diagonal and set all remaining nonzero values to one so that the resulting matrix is the adjacency matrix of an undirected graph. The properties of the resulting graphs are presented in Table 5.3.

As discussed in Section 5.4.2, getting rid of the entries that do not belong to any perfect matching is a more time consuming procedure than its equivalent for bipartite graphs. As Fürer and Kasiviswanathan [49] discard all edges that cannot be put into a perfect matching from the graph in the original description of GreedyMtc, we implemented this cleaning for GreedyMtc. For uniformity with the previous section, we shall use RasmussenMtc instead of Simple to refer to the extension of Rasmussen's algorithm for undirected graphs. Here, we select a row and then we remove any of its entries that do not participate in any perfect matching, so that we always end up with a valid perfect matching. As for estScalingMtc, we opted to test the following two alternatives:

1. The results labeled estScalingMtc in the figures correspond to Algorithm 5.2. In this version, the Dulmage–Mendelsohn decomposition is used to ensure that the matrix has total support. Then we cleaned only the entries from the selected row.

2. The results labeled with estClearScalingMtc are obtained by discarding all edges that are not part of some perfect matching, using first the Dulmage–Mendelsohn decomposition, and then by exhaustively testing the remaining edges (due to Fact 5.1, the resulting matrix has total support). Then we proceed to scale the matrix to obtain the scaling matrix $\mathbf{R}$ and do the appropriate selections. This corresponds to the cleaning performed in GreedyMtc.

We do not know the actual number of perfect matchings on these graphs. As in the previous subsection, we plot the distribution (in logarithmic scale) of the results for 1000 trials. The

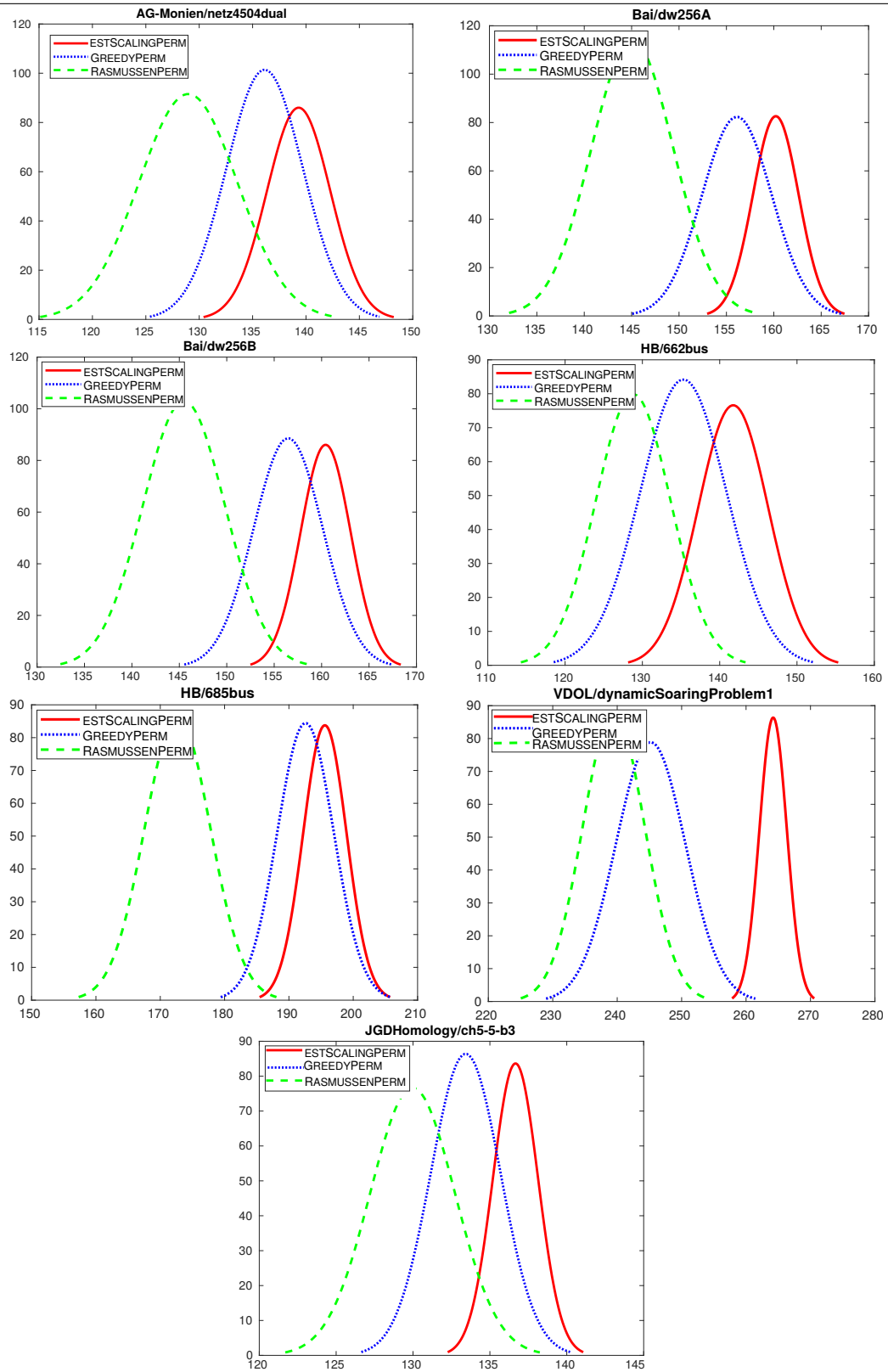


Figure 5.4 – The performance of the estimators on the seven matrices with $n > 500$ given in Table 5.1. The logarithms of the 1000 estimator values are presented on the x-axes and the y-axes demonstrate the distribution of the samples for the values in each approach. For these matrices, the permanent values are unknown.

Name	$\|V\|$	$\|E\|$
`Bai/bwm200`	100	298
`Bai/dw256A`	256	1004
`HB/ash292`	146	958
`JGDTrefethen/Trefethen200`	100	1345
`Pothen/mesh2em1`	153	856

Table 5.3 – The properties of undirected graphs corresponding to real-life matrices and the names of those matrices.

results are presented in Figure 5.5. We observe that the two scaling-based alternatives have similar curves, and there does not seem to be an advantage in applying the more expensive method of extensive cleaning. In addition, we observe that our approach seems to minimize the variance in respect to the other alternatives by providing a closer range of reported values.

Finally, for this set of experiments, we provide the means as well as the standard deviations in Table 5.4 using the same definitions as in the previous subsection. The table shows that once again the proposed algorithm obtains usually the smallest standard deviation to mean ratio. In addition, we can observe that the extensive cleaning of entries can help in reducing this ratio, though only slightly.

Focusing on the run time of the estimators, we have that again estScalingMtc is fast. In these graphs, RasmussenMtc algorithm requires on average 0.05 seconds to calculate an estimate for $M(G)$, whereas an estimator based on estScalingMtc requires 0.32 seconds on average. The other two methods GreedyMtc and estClearScalingMtc are both significantly slower. The increase in the run time is due to their excessive cleaning procedure which at east step eliminates all edges which do not participate in a perfect matching. GreedyMtc requires on average 8.68 seconds while estClearScalingMtc requires 8.92 seconds. Since estScalingMtc and estClearScalingMtc obtain similar results, we suggest avoiding the costly cleaning step of estClearScalingMtc.

We again note that the run time of estScalingMtc is greatly affected by the number of nonzeros as in the bipartite case. To improve its run time on larger instances we can similarly reduce the number of scaling iterations or use parallelism. Interestingly, unlike the bipartite case, the preprocessing step of GreedyMtc is costlier than that of estScalingMtc and thus GreedyMtc should remain slower than estScalingMtc in larger graphs as well.

5.6 Concluding remarks

In this chapter, we proposed an algorithm for approximating the number of perfect matchings in bipartite and general undirected graphs. The proposed algorithm uses matrix scaling in order to sample randomly a perfect matching of the graph. The proposed algorithm at each step selects a vertex and matches it randomly with one of its neighbors. The random decision is based on the values of the vertex's neighbors in the scaled adjacency matrix of the graph. At the end of the algorithm an estimate X is returned based on the values of the chosen probabilities. This process is repeated a sufficient number of times and a mean estimate $\overline{X}$ is returned by taking the mean of all returned X values. We proved loose yet computable upper bounds for the expected value of the square of X which affects the number of samples required. The experimental analysis demonstrated improvements over previous similar methodologies. Future work involves

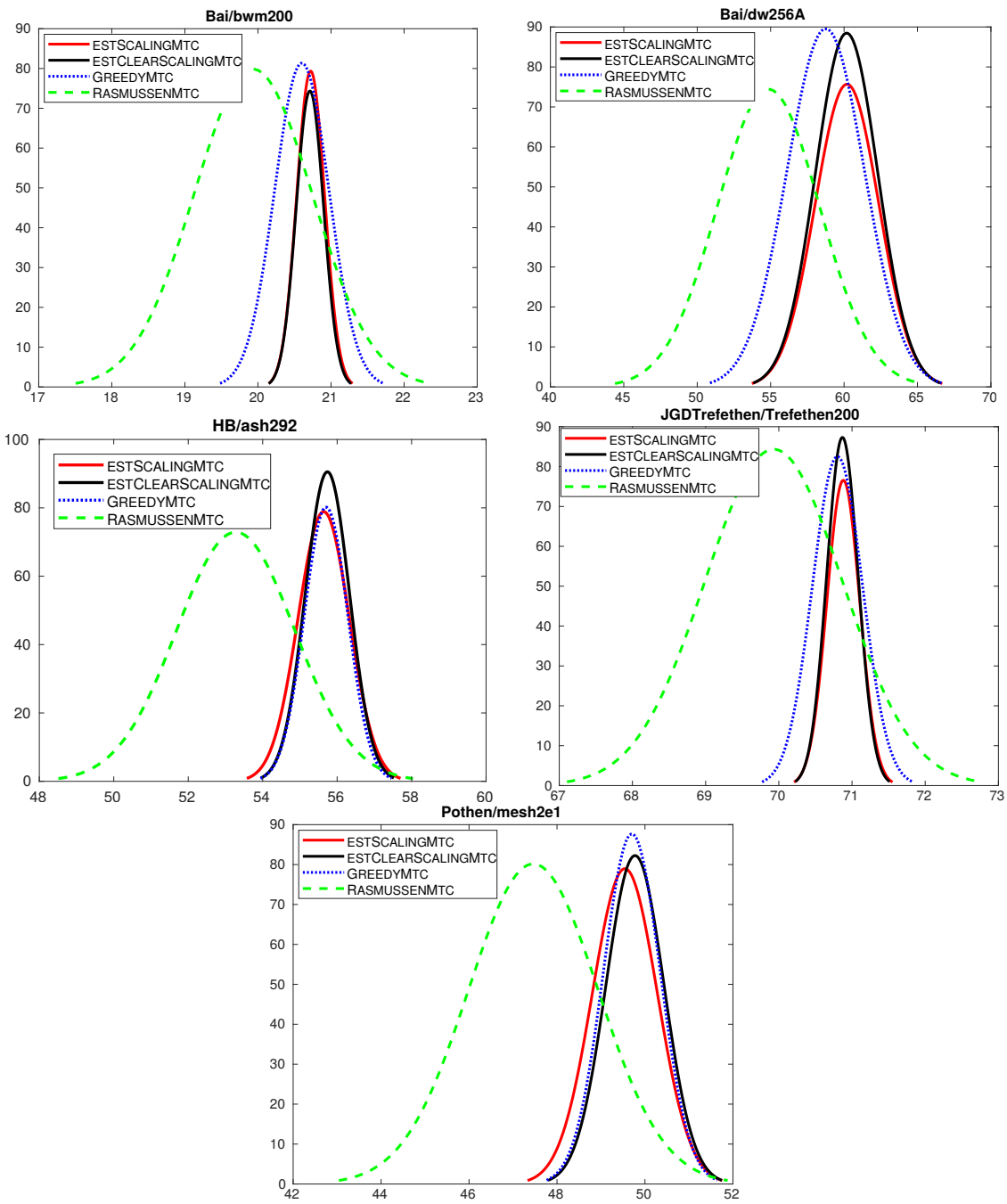


Figure 5.5 – The performance of the estimators on the five undirected graphs with $n < 260$ in Table 5.3. The logarithms of the 1000 estimator values are presented on the x-axes and the y-axes demonstrate the distribution of the samples for the values in each approach. For these graphs, the number of perfect matchings is unknown.

	RasmussenMtc			GreedyMtc		
Matrix	mean	std	$\frac{std}{mean}$	mean	std	$\frac{std}{mean}$
`Bai/bwm200`	$5.44 \cdot 10^{20}$	$2.71 \cdot 10^{21}$	4.98	$5.74 \cdot 10^{20}$	$5.84 \cdot 10^{20}$	1.02
`Bai/dw256A`	$1.71 \cdot 10^{61}$	$3.26 \cdot 10^{62}$	19.10	$2.58 \cdot 10^{62}$	$2.47 \cdot 10^{63}$	9.59
`HB/ash292`	$7.23 \cdot 10^{55}$	$8.73 \cdot 10^{56}$	12.07	$1.22 \cdot 10^{56}$	$2.26 \cdot 10^{56}$	1.86
`JGDTrefethen/Trefethen200`	$8.33 \cdot 10^{70}$	$4.14 \cdot 10^{71}$	4.98	$8.63 \cdot 10^{70}$	$7.81 \cdot 10^{70}$	0.90
`Pothen/mesh2e1`	$7.21 \cdot 10^{49}$	$1.03 \cdot 10^{51}$	14.35	$1.45 \cdot 10^{50}$	$3.80 \cdot 10^{50}$	2.63
	estScalingMtc			estClearScalingMtc		
	mean	std	$\frac{std}{mean}$	mean	std	$\frac{std}{mean}$
`Bai/bwm200`	$5.81 \cdot 10^{20}$	$2.81 \cdot 10^{20}$	0.48	$5.65 \cdot 10^{20}$	$2.62 \cdot 10^{20}$	0.46
`Bai/dw256A`	$6.88 \cdot 10^{62}$	$4.87 \cdot 10^{63}$	7.08	$3.16 \cdot 10^{62}$	$2.05 \cdot 10^{63}$	6.49
`HB/ash292`	$1.43 \cdot 10^{56}$	$3.66 \cdot 10^{56}$	2.56	$1.32 \cdot 10^{56}$	$2.77 \cdot 10^{56}$	2.10
`JGDTrefethen//Trefethen200`	$8.63 \cdot 10^{70}$	$4.44 \cdot 10^{70}$	0.51	$8.32 \cdot 10^{70}$	$4.11 \cdot 10^{70}$	0.49
`Pothen/mesh2e1`	$1.38 \cdot 10^{50}$	$3.14 \cdot 10^{50}$	2.27	$1.85 \cdot 10^{50}$	$5.20 \cdot 10^{50}$	2.81

Table 5.4 – Statistics for the five undirected graphs obtained from the matrices in Table 5.3.

bounding the estimates for special graph classes.

Chapter 6

Results on the Birkhoff–von Neumann decomposition

In this chapter we examine the Min-BvN-Decomp [39] problem which was defined in Section 1.3 of Chapter 1. The results of this chapter were published in Linear algebra and its applications [J1]. Recall from Section 1.3 that a permutation matrix is a square matrix such that each row and each column contain exactly one nonzero value equal to 1. Then, a doubly stochastic matrix can be written as a linear combination of permutation matrices $\mathbf{P}_1, \ldots, \mathbf{P}_k$ with positive coefficients $\alpha_1, \ldots, \alpha_k$ as

$$\mathbf{A} = \alpha_1 \mathbf{P}_1 + \alpha_2 \mathbf{P}_2 + \cdots + \alpha_k \mathbf{P}_k, \text{ where } \sum_{i=1}^{k} \alpha_i = 1 \ . \tag{6.1}$$

Such a representation is called the Birkhoff–von Neumann (BvN) [13] decomposition and is generally not unique. The NP-Hard Min-BvN-Decomp problem [39] then asks for a decomposition with the least number of permutations used.

We consider two heuristics called Birkhoff and $\text{Greedy}_{\text{BVN}}$ which can be used to obtain a BvN decomposition. The former is based on a constructive proof of Equation 6.1, while the latter greedily tries to maximize the values of the α coefficients. At first, we show that the theoretical guarantee of the Birkhoff heuristic can be arbitrarily bad. To achieve this we describe a family of matrices such that for an $n \times n$ matrix Birkhoff can return n permutation matrices, while the optimal answer is only 3. Kulkarni et al. [83, Theorem 7] present a family of matrices where Birkhoff obtains 2^ℓ permutation matrices while the optimal decomposition requires ℓ. Compared to their method, our construction is simpler, more explicit, and yields a stronger result. Our second contribution is an a posteriori approximation guarantee of the $\text{Greedy}_{\text{BVN}}$ heuristic. We present theoretical and experimental results to showcase how the performance of $\text{Greedy}_{\text{BVN}}$ is affected by the nonzero entries of a doubly stochastic matrix. Our third contribution finally modifies the $\text{Greedy}_{\text{BVN}}$ heuristic with an additional step. The conducted experimental analysis shows that the proposed modification is promising and can lead to decompositions with a smaller number of permutation matrices on several matrices.

The rest of the chapter is organized as follows. Section 6.1 introduces the two examined heuristics Birkhoff and $\text{Greedy}_{\text{BVN}}$ and in Section 6.2 we present our analysis of these two heuristics. Our modifications of the $\text{Greedy}_{\text{BVN}}$ heuristic are described in Section 6.3 and some experiments with them are given in Section 6.4. Section 6.5 summarizes the chapter and discusses some potential future work.

6.1 The two heuristics

There exist heuristics to compute a BvN decomposition for a given matrix $\mathbf{A}$. In particular, the following family of heuristics is based on the constructive process in Birkhoff's proof. 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 $\mathbf{A}^{(j-1)}$, denote 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 computation for the next step with $j = j + 1$ until $\mathbf{A}^{(j)}$ becomes the void matrix $\mathbf{0}$ where all the entries are equal to zero. Note that each $\mathbf{A}^{(j)}$ matrix has the same row and column sum, because at every step the same value is subtracted from each row and column. Any heuristic of this type is a member of the *generalized Birkhoff* family of heuristics. The overall procedure is summarized in Algorithm 6.1. To find a permutation at Line 5 one can employ a matching algorithm on the bipartite graph defined by $\mathbf{A}^{(j-1)}$. The two heuristics that we will examine and analyse belong to this family of heuristics.

Birkhoff: It is the original heuristic used to prove that a doubly stochastic matrix has a BvN decomposition. Let α be the position of the minimum entry in $\mathbf{A}^{(j-1)}$. Then the permutation $\mathbf{P}_j$ selected at the jth step contains α, i.e., $\alpha_j = a$.

Greedy$_{\text{BVN}}$ **[39]**: This heuristic selects the permutation that maximizes the value of the minimum element. To find a permutation with this property, one can consider the *maximum bottleneck matching* problem [50] in the bipartite graph defined by the matrix $\mathbf{A}^{(j-1)}$. Greedy$_{\text{BVN}}$ maximizes α_j and therefore α_j is the largest amount which can be subtracted from a row and column of $\mathbf{A}^{(j-1)}$. This heuristic hence aspires to make $\mathbf{A}^{(j)}$ void as soon as possible by always performing the largest possible reductions.

Algorithm 6.1: Generalized-Birkhoff: Template to find a BvN decomposition

Input: A doubly stochastic matrix $\mathbf{A}$.
Output: A valid BvN decomposition of $\mathbf{A}$.

1: $\mathbf{A}^{(0)} \leftarrow \mathbf{A}$
2: $j \leftarrow 0$
3: **while** $\mathbf{A}^{(j)} \neq \mathbf{0}$ **do**
4: $\quad j \leftarrow j + 1$
5: $\quad \mathbf{P}^{(j)} \leftarrow$ a permutation matrix in the pattern of $\mathbf{A}^{(j-1)}$
6: $\quad \alpha_j \leftarrow$ the minimum value in $\mathbf{A}^{(j-1)}$ at the nonzero positions of $\mathbf{P}^{(j-1)}$
7: $\quad \mathbf{A}^{(j)} \leftarrow \mathbf{A}^{(j-1)} - \alpha^{(j)} \cdot \mathbf{P}^{(j)}$

6.2 Analysis of the two heuristics for computing BvN decompositions

An empirical study from the literature [39] demonstrated that the Greedy$_{\text{BVN}}$ heuristic obtains a much smaller number of permutation matrices compared to the Birkhoff heuristic in practice. Here, we compare these two heuristics theoretically in an attempt to explain this behavior. First we show that the original Birkhoff heuristic does not have any constant ratio approximation guarantee. For an $n \times n$ matrix, its worst-case approximation ratio is $\Omega(n)$.

We begin with a small example shown in Figure 6.1. We decompose a 6×6 matrix $\mathbf{A}^{(0)}$ which has an optimal BvN decomposition with three permutation matrices: the main diagonal,

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

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

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

Figure 6.1 – A sample matrix to show that the original BIRKHOFF heuristic can obtain a BvN decomposition with n permutation matrices while the optimal one has 3.

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 $\mathbf{A}^{(0)}$ are equal to 6, it can be converted to a doubly stochastic matrix by dividing all the entries with 6. The figure shows how one can obtain a suboptimal decomposition with 6 matrices using the BIRKHOFF heuristic. In all steps, the red-colored entry set corresponds to a permutation and contains the minimum possible value in the matrix, which is 1. In the following, we show how to generalize the idea behind Figure 6.1 to matrices of arbitrary size.

Lemma 6.1. *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 can obtain a BvN decomposition with exactly n permutation matrices. The example in Figure 6.1 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 is $f(\cdot)$ 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

$$\overbrace{(1,1)}^{\text{1st set}}, \overbrace{(n,2)}^{\text{2nd set}}, \overbrace{(2,3),(3,4),\ldots,(n-1,n)}^{\text{3rd set}} . \tag{6.2}$$

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 (6.2). 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 (6.2). 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 (6.2). 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 the approximation for Birkhoff is no better than $\frac{n}{3}$, which can be made arbitrarily large. □

Note that $\text{Greedy}_{\text{BVN}}$ will optimally decompose the matrix $\mathbf{A}$ used in the above proof. We now analyze the performance of the $\text{Greedy}_{\text{BVN}}$ heuristic. Initially, we present some experiments showcasing that there is a strong connection between the performance of $\text{Greedy}_{\text{BVN}}$ and the values in the matrix.

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

$n = 30$ and $z = 20$					$n = 200$ and $z = 100$				
	average		worst case			average		worst case	
i	k_i	k_i/z	k_i	k_i/z	i	k_i	k_i/z	k_i	k_i/z
10	59	2.99	63	3.15	10	268	2.69	280	2.80
20	105	5.29	110	5.50	20	487	4.88	499	4.99
30	149	7.46	158	7.90	30	716	7.16	726	7.26
40	184	9.23	191	9.55	40	932	9.33	947	9.47
50	212	10.62	227	11.35	50	1124	11.25	1162	11.62

Table 6.1 – Experiments showing the dependence of the performance of $\textsc{Greedy}_{\text{BVN}}$ 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 $\mathbf{A}_i$. We did five experiments for a given pair of n and i. $\textsc{Greedy}_{\text{BVN}}$ obtains k_i permutation matrices. The average and the maximum number of permutation matrices obtained by $\textsc{Greedy}_{\text{BVN}}$ for five random instances are given. $\frac{k_i}{z}$ is a lower bound to the performance of $\textsc{Greedy}_{\text{BVN}}$, as $z \geq Opt$.

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 in Figure 6.1, each $\mathbf{A}_i$ is not doubly stochastic but it has equal row and columns sums and can be easily turned into one. 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 some results in Table 6.1 for two different values of n and z. We chose $n = 30$ and $z = 20$ for the first set, and $n = 200$ and $z = 100$ for the second. We created five different set of permutations $\{\mathbf{C}_1, \ldots, \mathbf{C}_z\}$ and tested for $i \in \{10, 20, 30, 40, 50\}$. Let k_i be the number of permutation matrices that $\textsc{Greedy}_{\text{BVN}}$ obtains for a given $\mathbf{A}_i$. The table gives the average and the maximum k_i of the five different $\mathbf{A}_i$ matrices, for a given pair of n and i. By construction, each $\mathbf{A}_i$ has a BvN decomposition 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 $\textsc{Greedy}_{\text{BVN}}$. As seen from the experiments, as i increases, the performance of $\textsc{Greedy}_{\text{BVN}}$ gets increasingly worse. This shows that a constant ratio worst case approximation of $\textsc{Greedy}_{\text{BVN}}$ is unlikely. While the performance depends on z (for example, for small z, $\textsc{Greedy}_{\text{BVN}}$ is likely to obtain near optimal decompositions), it seems that the size of the matrix does not largely affect the relative performance of $\textsc{Greedy}_{\text{BVN}}$. Now we attempt to explain the above results theoretically.

Lemma 6.2. *Let $\alpha_1^\star \mathbf{P}_1^\star + \cdots + \alpha_k^\star \mathbf{P}_{k^\star}^\star$ be an optimal BvN decomposition of a given doubly stochastic matrix $\mathbf{A}$. 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 not have $\max_i \alpha_i^\star$. Luckily, we can say more in the case of $\textsc{Greedy}_{\text{BVN}}$.

Corollary 6.1. *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 $\textsc{Greedy}_{\text{BVN}}$ heuristic for decomposing $\mathbf{A}$. Then, $\ell \leq k^\star \cdot \frac{\alpha_1}{\alpha_\ell}$.*

Proof. This is easy to see as $\textsc{Greedy}_{\text{BVN}}$ obtains the coefficients in a non-increasing order [39, Lemma 3], and $\alpha_1 \geq \alpha_j^\star$ for all $1 \leq j \leq k^\star$ for any BvN decomposition containing $\alpha_j^\star$. □

Lemma 6.2 and Corollary 6.1 give a posteriori estimates of the performance of the $\textsc{Greedy}_{\text{BVN}}$ heuristic, in that one looks at the decomposition and tells how good it is. This potentially can reveal a good performance. For example, when $\textsc{Greedy}_{\text{BVN}}$ 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 of Figure 6.1). We also note that the ratio given in Corollary 6.1 should usually be much larger than the practical performance.

6.3 Heuristics for $\textsc{Greedy}_{\text{BVN}}$

As was discussed above, the $\textsc{Greedy}_{\text{BVN}}$ algorithm operates by trying to maximize the subtracted value and consequently minimize the column/row sum in the resulting matrix. This is done in the belief that by reducing maximally, the resulting matrix will require less steps in order to be fully decomposed. However, a matrix can contain several distinct permutations with the same maximum bottleneck value. Each one of those equivalent (in terms of the bottleneck value) permutations leads to a different matrix in the following steps and hence a different number of permutation matrices. Consider for example two possible executions of the $\textsc{Greedy}_{\text{BVN}}$ heuristic for the following 4×4 matrix

$$\begin{pmatrix} 1 & 5 & 0 & 4 \\ 2 & 4 & 0 & 4 \\ 3 & 1 & 6 & 0 \\ 4 & 0 & 4 & 2 \end{pmatrix} \ .$$

The first execution returns an optimal solution with $k = 4$ using the following decomposition

$$4\begin{pmatrix}0&0&0&1\\0&1&0&0\\0&0&1&0\\1&0&0&0\end{pmatrix}+3\begin{pmatrix}0&1&0&0\\0&0&0&1\\1&0&0&0\\0&0&1&0\end{pmatrix}+2\begin{pmatrix}0&1&0&0\\1&0&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}+1\begin{pmatrix}1&0&0&0\\0&0&0&1\\0&1&0&0\\0&0&1&0\end{pmatrix}.$$

However, for the second execution we can obtain $k = 5$ by choosing different permutation matrices in the first and the second steps (with the equivalent bottleneck values as in the previous execution)

$$4\begin{pmatrix}0&1&0&0\\0&0&0&1\\0&0&1&0\\1&0&0&0\end{pmatrix}+3\begin{pmatrix}0&0&0&1\\0&1&0&0\\1&0&0&0\\0&0&1&0\end{pmatrix}+1\begin{pmatrix}0&1&0&0\\1&0&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}+1\begin{pmatrix}0&0&0&1\\1&0&0&0\\0&1&0&0\\0&0&1&0\end{pmatrix}+1\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}.$$

This led us to investigate whether or not some of the permutations have better properties than others and what other secondary criteria we could take into account while choosing a permutation for $\textsc{Greedy}_{\text{BVN}}$. At the end, we considered a variation of $\textsc{Greedy}_{\text{BVN}}$ called $\textsc{Greedy}^{+}_{\text{BVN}}$. $\textsc{Greedy}^{+}_{\text{BVN}}$ returns a maximum bottleneck matching that maximizes the sum of its entries. As the experiments of Table 6.1 showed, $\textsc{Greedy}_{\text{BVN}}$ behaves better when the values in the matrix are relatively small. In finding the maximum sum, we are optimistic that such a permutation will contain a lot of large elements which will then be reduced maximally by the maximum bottleneck value for $\mathbf{A}$.

The algorithm implementing $\textsc{Greedy}^{+}_{\text{BVN}}$ requires two steps. In the first step, as in the $\textsc{Greedy}_{\text{BVN}}$ heuristic, we apply a maximum bottleneck matching algorithm on $\mathbf{A}$ to find the bottleneck value b. We then create a new matrix $\mathbf{A}_b$ keeping only those values from $\mathbf{A}$ such that $\alpha_{ij} \geq b$. In the second step, a permutation of $\mathbf{A}_b$ with maximum weight is returned. Since $\mathbf{A}_b$ comprises only of values from $\mathbf{A}$, the returned permutation is also a permutation for $\mathbf{A}$ and because any value is at least as large as b it is also a maximum bottleneck matching. Understandably, $\textsc{Greedy}^{+}_{\text{BVN}}$ requires more computational time than $\textsc{Greedy}_{\text{BVN}}$ as an additional maximum weighted perfect matching algorithm needs to be run on top of the maximum bottleneck matching algorithm at each step.

We additionally consider a relaxation of $\textsc{Greedy}^{+}_{\text{BVN}}$ which we refer to as $\textsc{Greedy}^{R}_{\text{BVN}}$. This relaxation decides whether to apply the secondary maximum weight sum algorithm or not based on the outcome of a biased coin. More specifically, at each step, with probability p, $\textsc{Greedy}_{\text{BVN}}$ is chosen and with the complement probability $1-p$ we opt instead for the $\textsc{Greedy}^{+}_{\text{BVN}}$ variant. In doing so, we hope to combine the best elements of both $\textsc{Greedy}_{\text{BVN}}$ and $\textsc{Greedy}^{+}_{\text{BVN}}$ and construct a heuristic that would provide satisfactory answers for a larger set of matrices.

6.4 Experiments on real-life matrices

To see the effects of the two new proposed heuristics we conducted experiments with 55 real-life sparse matrices taken from the SuiteSparse Matrix Collection [31]. We used MATLAB interfaces for the software MC64 [36, 37] which provides codes written in FORTRAN for the maximum bottleneck matching and maximum weighted sum matching problems.

We preprocessed these matrices with matrix scaling algorithms [79, 80] to make them (nearly)

name	n	$\textsc{Greedy}_{\text{BVN}}$	$\textsc{Greedy}^{+}_{\text{BVN}}$	$\textsc{Greedy}^{R}_{\text{BVN}}$
3elt_dual	9000	310	281	298
aft01	8205	120	122	119
aft02	8184	228	216	231
airfoil1_dual	8034	332	279	309
barth	6691	71	71	66
bcsstk33	8738	468	494	459
bcsstk38	8032	589	539	572
benzene	8219	112	119	114
c-29	5033	864	851	830
c-36	7479	1718	1361	1594
c-37	8204	889	761	818
EX5	6545	228	221	225
EX6	6545	228	220	236
ex40	7740	394	311	313
fxm3_6	5026	370	368	360
igbt3	10938	2001	1155	1732
Kuu	7102	327	306	304
nemeth01	9506	207	219	205
pf2177	9728	993	1069	946
pkustk02	10800	615	645	591
s2rmq4m1	5489	207	206	215
SiH4	5041	567	451	534
ted_AB	10605	380	361	374

Table 6.2 – The number of permutation matrices in a BvN decomposition obtained by the three heuristics for a subset of 23 matrices from all 55 matrices. Matrix `igbt3` could not be decomposed with $\textsc{Greedy}_{\text{BVN}}$ within 2000 iterations that is why its k exceeds this limit.

doubly stochastic. Prior to scaling, we converted any negative values in these matrices to positive, since nonnegative entries are a prerequisite for doubly stochastic matrices.

We included some stopping conditions to the decomposition process. As a first condition, we set a limit of 2000 iterations for each heuristic. Once this limit has been reached, we stop the procedure at that point even if the matrix has not been decomposed fully yet. This condition very rarely occurred in the tests.

The other conditions take care of the issues that are created by working with floating point numbers. Due to the fact that FP-arithmetic is not exact, it might be impossible to make $\mathbf{A}^{(i)}$ consist only of exact zeros. Instead a threshold θ is defined and if some value α_{ij} ends up being below θ, it is considered equal to zero and set as such. Finally, we stop if the accumulated sum of the coefficients (that is, $\sum \alpha_i$) is nearly equivalent to 1, i.e., over 0.9999. We tested $\textsc{Greedy}^{R}_{\text{BVN}}$ with $p = 0.6$, i.e., 60% of the time it applied the simple $\textsc{Greedy}_{\text{BVN}}$ heuristic.

We present some of the results in Tables 6.2 to 6.4. Focusing solely on the performance of $\textsc{Greedy}^{+}_{\text{BVN}}$ in Table 6.2 we can make a few observations. First of all, our augmentation seems indeed to influence positively the quality of the returned solution. $\textsc{Greedy}^{+}_{\text{BVN}}$ manages to outperform $\textsc{Greedy}_{\text{BVN}}$ in many instances. It is particularly of note to examine the instance `igbt3`. As we can see the original $\textsc{Greedy}_{\text{BVN}}$ finished without being able to fully decompose this matrix whereas $\textsc{Greedy}^{+}_{\text{BVN}}$ decomposed the matrix after the 1155th step. This signals an improvement of at least 43% which could be higher since we did not run $\textsc{Greedy}_{\text{BVN}}$ to completion. There also exist some matrices where $\textsc{Greedy}^{+}_{\text{BVN}}$ either returned an inferior re-

Heuristic	$\textsc{Greedy}_{\text{BVN}}$ $<$	$\leq$
$\textsc{Greedy}^{+}_{\text{BVN}}$	26	45
$\textsc{Greedy}^{R}_{\text{BVN}}$	37	47

Table 6.3 – We present the number of experiments for which the two heuristics obtained a smaller number of permutations than $\textsc{Greedy}_{\text{BVN}}$ (first column) or the same number of permutations (second column). We performed in total 55 tests.

Heuristic	k
$\textsc{Greedy}_{\text{BVN}}$	418
$\textsc{Greedy}^{+}_{\text{BVN}}$	388
$\textsc{Greedy}^{R}_{\text{BVN}}$	403

Table 6.4 – Average number of decompositions used over all experiments

sult to $\textsc{Greedy}_{\text{BVN}}$ or failed to improve $\textsc{Greedy}_{\text{BVN}}$'s output. This is perhaps unavoidable as $\textsc{Greedy}_{\text{BVN}}$ has less restrictions on which permutation to pick and thus potentially can pick a good one more frequently in some instances. It is in such cases, that the proposed relaxation $\textsc{Greedy}^{R}_{\text{BVN}}$ becomes useful. As can be seen in Table 6.3, $\textsc{Greedy}^{R}_{\text{BVN}}$ manages to decompose matrices with less permutations than $\textsc{Greedy}_{\text{BVN}}$ more frequently than $\textsc{Greedy}^{+}_{\text{BVN}}$ (first column), while in overall it is at least as good as $\textsc{Greedy}_{\text{BVN}}$ in more instances (second column). Quality-wise, the variant $\textsc{Greedy}^{R}_{\text{BVN}}$ is not always as good as $\textsc{Greedy}^{+}_{\text{BVN}}$ (see the average numbers in Table 6.4 and the reported numbers in Table 6.2). It instead seems to offer a more balanced approach and succeeded at beating $\textsc{Greedy}_{\text{BVN}}$ in more cases.

6.5 Concluding remarks

In this chapter, we investigated heuristics for obtaining Birkhoff–von Neumann (BvN) decomposition of doubly stochastic matrices and presented three results. First, we showed that the worst-case approximation ratio of the original Birkhoff heuristic is $\Omega(n)$. Second, we showed how the performance of the $\textsc{Greedy}_{\text{BVN}}$ heuristic depends on the values of matrix elements, and obtained an a posteriori bound for $\textsc{Greedy}_{\text{BVN}}$ using the first and the last coefficients in its obtained decomposition. The shown bound for the performance of $\textsc{Greedy}_{\text{BVN}}$ 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 $\textsc{Greedy}_{\text{BVN}}$ heuristic (which was demonstrated earlier [39]). Third, we discussed two modifications of the $\textsc{Greedy}_{\text{BVN}}$ heuristic to reduce the number of permutations in the produced BvN decomposition of a doubly stochastic matrix. On this theme, we are interested in additional modifications to the $\textsc{Greedy}_{\text{BVN}}$ algorithm. It would also be interesting to apply the degree-1 and degree-2 reduction rules for the maximum weighted matching [82] as a means to improve the run time of both $\textsc{Greedy}^{+}_{\text{BVN}}$ and $\textsc{Greedy}^{R}_{\text{BVN}}$.

Another possible avenue for research comes from the constructive proof of the Birkhoff–von Neumann theorem due to Koopmans and Beckmann [81]. In this approach, $\mathbf{A}$ is recursively

split into two matrices $\mathbf{B}$ and $\mathbf{C}$ such that $\mathbf{A} = \beta\mathbf{B} + (1-\beta)\mathbf{C}$, and $0 < \beta < 1$. A BvN decomposition of $\mathbf{A}$ can be obtained by combining the BvN decompositions of $\mathbf{B}$ and $\mathbf{C}$. In short, their proposed method starts by finding a cycle. Then, the matrices $\mathbf{B}$ and $\mathbf{C}$ are created by deleting from this cycle the even numbered edge of lowest weight (for $\mathbf{B}$) or the odd numbered edge of lowest weight (for $\mathbf{C}$). The remaining edges in the cycles are updated appropriately (and in a different manner) in both $\mathbf{B}$ and $\mathbf{C}$. The β parameter is defined based on the two values in the two deleted edges. We refer the reader to the original paper for a more detailed overview.

A direct application of the above yields an exponential time algorithm. The matrices $\mathbf{B}$ and $\mathbf{C}$ overlap significantly and as a consequence share many permutations, which leads to combinatorially large space requirements and/or very high run time. Unlike heuristics from the generalized Birkhoff heuristic family, an approach based on the algorithm by Koopmans and Beckmann can potentially find BvN decompositions which require a much higher number of permutation matrices than the aforementioned upper bounds, i.e., $\tau - 2n + 2$. On the other hand, Uçar [105] shows that such an approach can be guided to find optimal decompositions in instances where any possible heuristic following the template of Algorithm 6.1 cannot. Attempting to make the algorithm by Koopmans and Beckmann useful in practice can be therefore important.

Chapter 7

Conclusion

In this thesis we examined four important problems in the field of combinatorial scientific computing. These problems were that of finding matchings in graphs and hypergraphs, the estimation of the number of perfect matchings in graphs, and the Birkhoff–von Neumann decomposition of doubly stochastic matrices. Our main contribution in the works which constitute this thesis was to examine the relation between matching and matrix scaling. The proposed algorithms in Chapters 2–5, which make use of scaling, exhibited the best performance in practice versus the standard approaches from the literature. Chapter 6 worked on doubly stochastic matrices, where those matrices could potentially be coming from the scaling of general matrices [9]. Before discussing potential future directions for research we give a brief summary of each individual chapter.

7.1 Summary of the chapters

In Chapter 2 we examined the matching problem in bipartite graphs, where the problem is simpler due to the absence of odd cycles. This chapter consisted of three parts. In the first part, we studied in detail the well-known matching heuristic by Karp and Sipser [72]. We proposed a subquadratic implementation of the heuristic with an expected $\mathcal{O}(m \log n)$ complexity, which is currently the best known. In the second part of the chapter, we considered two exact probabilistic matching algorithms for two special classes of bipartite graphs [54, 71]. Using matrix scaling, we converted these two specialized algorithms into two practical heuristics called 2OUTMC and TRUNCRW that are applicable to arbitrary bipartite graphs. The conducted experiments showed that these two heuristics are fast and obtain near-optimal matchings in several real-life or synthetic graphs. They are also robust and very effective as initialization methods. In addition, we empirically showed that k-out subgraphs from general bipartite host graphs for $k = 2, 3$ either had a perfect matching or only left a few vertices unmatched. For $k = 3$, in particular, a perfect matching was found in all cases. Considering that k-out subgraphs have $\mathcal{O}(n)$ edges, this observation seems to suggest that an $\mathcal{O}(n\sqrt{n})$ algorithm should return either a maximum matching or a matching that requires only minimal work to be made perfect. Finally, in the last part of the chapter we proposed a deterministic heuristic for matching. This heuristic was obtained through the derandomization of an existing matching heuristic [38] based on matrix scaling. We presented some experiments showing that the deterministic heuristic performs consistently better than its current $1 - 1/e$ approximation guarantee.

Chapter 3 examined the maximum matching problem in general undirected graphs. We adapted an earlier algorithm [38] for bipartite graphs and discussed some variations of the main

algorithm. The adapted algorithm works by creating a random 1-out subgraph of the host graph. We showed that the algorithm has an approximation guarantee of around $0.866 - \log(n)/n$ for complete graphs. We focused on complete graphs as their equivalents on bipartite graphs lead to the worst behavior for the original algorithm [38] that we adapted. The experiments verified the good performance of the heuristic.

In Chapter 4 we proposed heuristics for the NP-Hard problem of finding a maximum cardinality matching in a d-partite, d-uniform hypergraph. These heuristics started as generalizations of existing matching heuristics for graphs. We then proceeded to propose tensor scaling-based heuristics inspired by the works in the previous two chapters. Our experiments showcased that the heuristics based on scaling had the best performance in a large set of tests.

In Chapter 5 we investigated randomized methods for approximating the number of perfect matchings in bipartite graphs and general undirected graphs. The proposed algorithms and their analysis rely on matrix scaling. We provided a sufficient upper bound to guarantee an (ε, δ)-approximation for given choice of ε and δ using the scaling coefficients of the initial adjacency matrix of the graph. We then showed in practice that the scaling-based methods return the most accurate approximations versus two other alternatives based on similar methodology.

In Chapter 6 we examined two heuristics for obtaining a BvN decomposition with a small number of permutation matrices. We showed performance bounds for both of these heuristics. We then discussed an extension of one of these heuristics for improved performance.

7.2 Future work

We recall and collect some of the future work discussed at the end of each chapter to give further insights.

In the first part of Chapter 2, we left open the question of whether a lower complexity implementation of Karp–Sipser is possible or not. Rules similar to Rule-2 of Karp–Sipser are often used as reduction techniques in NP-Complete problems such as Maximum-Independent-Set [84] or Vertex-Cover [42]. A linear or pseudo-linear algorithm to apply Rule-2 and efficiently merge the neighbors of degree-2 vertices should therefore be of interest to several problems.

Our findings in Chapter 2 also necessitate the further examination of k-out bipartite graphs sampled from a host graph based on matrix scaling. In particular, our aim in this regard would be derive a theorem similar to the one by Walkup [107] about k-out graphs having a perfect matching. On this note, we also wonder whether 2outMC can be made exact for 2-out subgraphs of arbitrary host graphs. As we discussed in Section 2.3.1, a fast exact algorithm for matching in 2-out subgraphs can be used to yield an exact algorithm for arbitrary bipartite graphs with potentially reduced run time than the best known approaches [62].

We are also interested in derandomizing other randomized algorithms with higher approximation guarantees. It seems fitting to start this attempt with the TwoSided [38] heuristic which generates an 1-out subgraph having an 0.866-approximation of the maximum matching for a given bipartite graph. On this note, we plan to re-evaluate our analysis of the derandomized heuristic in an attempt to explain better the performance seen in the experiments.

A more involved research direction along the lines of Chapter 2 is to develop heuristics for the maximum weighted sum matching problem based on matrix scaling. Basing the probabilistic decisions solely on the values of the doubly stochastic matrix in a straightforward adaptation of our heuristics will not work. In addition to the original weighted bipartite graph, another weighted bipartite graph is created corresponding to the scaled adjacency matrix. The ordering

of the entries in a matrix $\mathbf{A}$ is not necessarily kept in the scaled matrix $\mathbf{S}$, i.e., $a_{ij} \geq a_{ij'}$ does not imply that $s_{ij} \geq s_{ij'}$. Thus while the two graphs have the same set of edges, the matchings which obtain the maximum weighted sum can be different. An approximation guarantee is therefore not necessarily maintained while passing from one graph to the other. This should thus be taken into consideration when trying to define an algorithm utilizing matrix scaling. As a first step, one could try to consider a related problem about maximizing the product of the weights in the obtained matching instead of the sum. There, the permutations which yield the maximum product are the same in both graphs, and it is potentially easier to adapt some of the results.

In Chapter 3 our analysis of the main algorithm was complicated by the existence of odd cycles in the generated subgraph. Empirically we showed that the obtained bound on complete graphs should be valid for general undirected graphs. We plan to re-examine our analysis and attempt to show that the $\Theta(\log n/n)$ bound on the number of odd cycles also holds theoretically for graphs whose adjacency matrices have total support. We also plan to extend both 2OUTMC and TRUNCRW to undirected graphs and study their behavior. As we discussed in the corresponding chapter, the adaptation of 2OUTMC has some challenges because 2OUTMC takes advantage of the bipartiteness of the given graph.

The future work concerning Chapter 4 bares similarities with the future work mentioned in the previous paragraphs, as our work in this chapter was inspired by our results for graphs. We are likewise interested in examining k-out hypergraphs and prove theoretically that a d^{d-1}-out hypergraph has a perfect matching. Furthermore, we would like to examine the weighted sum variant of the matching problem in d-partite, d-uniform hypergraphs. The same issues that complicate the extension of our results for bipartite graphs in the weighted case should also apply in here. We also intent to adapt our findings from the first part of Chapter 2 to improve our implementation of KARPSIPSERH. Avoiding the use of a stack to perform merges, in particular, should lead to a boost in performance.

An interesting long term perspective would be to design heuristics for the general problem of matching in hypergraphs, where the given hypergraphs are not necessarily d-partite and d-uniform. As a first step in this direction, we note that the two rules we proposed for KARPSIPSERH should be easily extendable for arbitrary hypergraphs.

Future work for Chapter 5 concerns mostly the re-examination of the conducted theoretical analysis. Tightening the obtained bounds should allow us to explain the good behavior that we observed in practice. In addition, we are interested in examining the behavior of the algorithm in certain graph classes such as grids and Erdös-Renyi graphs.

For the MIN-BVN-DECOMP problem discussed in Chapter 6, we are interested mostly in obtaining better performing heuristics. To that end, we intent to study the hardness of the problem and attempt to show if it is possible to obtain an algorithm with a constant factor approximation. Beyond that, we are going to examine the $\text{GREEDY}_{\text{BVN}}$ heuristic further in order to improve its performance, and potentially the performance of other related heuristics from the generalized Birkhoff family. We would also like to experiment with the method due to Koopmans and Beckmann [81] to see whether we could produce a practical algorithm out of it.

Bibliography

[1] IBM ILOG CPLEX Optimizer. https://www.ibm.com/analytics/cplex-optimizer. Last accessed: 07-06-2020.

[2] R. Aharoni and P. Haxell. Hall's theorem for hypergraphs. *Journal of Graph Theory*, 35(2):83–88, 2000.

[3] Z. Allen-Zhu, Y. Li, R. Oliveira, and A. Wigderson. Much faster algorithms for matrix scaling. *arXiv preprint arXiv:1704.02315*, 2017.

[4] M. Anastos and A. Frieze. Finding perfect matchings in random cubic graphs in linear time. *arXiv preprint arXiv:1808.00825*, 2018.

[5] J. Aronson, M. Dyer, A. Frieze, and S. Suen. Randomized greedy matching II. *Random Struct. Algor.*, 6(1):55–73, 1995.

[6] J. Aronson, A. M. Frieze, and B. G. Pittel. Maximum matchings in sparse random graphs: Karp-sipser revisited. *Random Struct. Algorithms*, 12(2):111–177, 1998.

[7] M. Bartha and M. Kresz. A depth-first algorithm to reduce graphs in linear time. In *11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 273–281, Sep. 2009.

[8] I. Beichl and F. Sullivan. Approximating the permanent via importance sampling with application to the dimer covering problem. *J. Comput. Phys.*, 149(1):128–147, 1999.

[9] M. Benzi and B. Uçar. Preconditioning techniques based on the Birkhoff–von Neumann decomposition. *Computational Methods in Applied Mathematics*, 17:201–215, 2017.

[10] C. Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the USA*, 43:842–844, 1957.

[11] P. Berman and M. Karpinski. Improved approximation lower bounds on small occurence optimization. *ECCC Report*, 2003.

[12] B. Besser and M. Poloczek. Greedy matching: Guarantees and limitations. *Algorithmica*, 77(1):201–234, 2017.

[13] G. Birkhoff. Tres observaciones sobre el algebra lineal. *Univ. Nac. Tucumán Rev. Ser. A*, (5):147–150, 1946.

[14] N. Blum. A new approach to maximum matching in general graphs. In *ICALP '90*, pages 586–597, London, UK, UK, 1990. Springer-Verlag.

[15] T. Bohmand and A. Frieze. Hamilton cycles in 3-out. *Random Structures & Algorithms*, 35(4):393–417, 2009.

[16] R. A. Brualdi. Notes on the Birkhoff algorithm for doubly stochastic matrices. *Canadian Mathematical Bulletin*, 25(2):191–199, 1982.

[17] R. A. Brualdi and P. 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.

[18] R. Burkard, M. Dell'Amico, and S. Martello. *Assignment problems, revised reprint*, volume 106. Siam, 2012.

[19] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr., and T. M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, volume 5, page 3, 2010.

[20] Ü. V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0*. Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. Available at https://www.cc.gatech.edu/~umit/software.html, 1999.

[21] D. Chakrabarty and S. Khanna. Better and simpler error analysis of the Sinkhorn-Knopp algorithm for matrix scaling. *arXiv preprint arXiv:1801.02790*, 2018.

[22] C. Chang, W. Chen, and H. 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.

[23] P. Chebolu, A. M. Frieze, and P. Melsted. Finding a maximum matching in a sparse random graph in $O(n)$ expected time. *J. ACM*, 57(4):24:1–24:27, 2010.

[24] E. Cohen. Structure prediction and computation of sparse matrix products. *Journal of Combinatorial Optimization*, 2(4):307–332, Dec 1998.

[25] M. B. Cohen, A. Madry, D. Tsipras, and A. Vladu. Matrix scaling and balancing via box constrained Newton's method and interior point methods. *arXiv preprint arXiv:1704.02310*, 2017.

[26] L. Cui, W. Li, and M. K. Ng. Birkhoff–von Neumann Theorem for multistochastic tensors. *SIAM Journal on Matrix Analysis and Applications*, 35(3):956–973, 2014.

[27] M. 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.

[28] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized algorithms*. Springer, 2015.

[29] M. Cygan, F. Grandoni, and M. 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.

[30] T. A. Davis. *Direct methods for sparse linear systems*. Siam, 2006.

[31] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.

[32] P. Devlin and J. Kahn. Perfect fractional matchings in k-out hypergraphs. *arXiv preprint arXiv:1703.03513*, 2017.

[33] I. S. Duff. On algorithms for obtaining a maximum transversal. *ACM Transactions on Mathematical Software*, 7(3):315–330, 1981.

[34] I. S. Duff, A. M. Erisman, and J. Reid. *Direct methods for sparse matrices.* Oxford University Press, 2017.

[35] I. S. Duff, K. Kaya, and B. Uçar. Design, implementation, and analysis of maximum transversal algorithms. *ACM Transactions on Mathematical Software*, 38(2):13, 2011.

[36] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 20(4):889–901, 1999.

[37] I. S. Duff and J. 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.

[38] F. Dufossé, K. Kaya, and B. Uçar. Two approximation algorithms for bipartite matching on multicore architectures. *J. Parallel Distr. Com.*, 85:62–78, 2015.

[39] F. Dufossé and B. Uçar. Notes on Birkhoff–von Neumann decomposition of doubly stochastic matrices. *Linear Algebra and its Applications*, 497:108–115, 2016.

[40] A. L. Dulmage and N. S. Mendelsohn. Coverings of bipartite graphs. *Canad. J. Math.*, 10:517–534, 1958.

[41] M. Dyer and A. Frieze. Randomized greedy matching. *Random Structures & Algorithms*, 2(1):29–45, 1991.

[42] M. R. Fellows, L. Jaffke, A. I. Király, F. A. Rosamond, and M. Weller. What is known about vertex cover kernelization? In *Adventures Between Lower Bounds and Higher Altitudes*, pages 330–356. Springer, 2018.

[43] T. I. Fenner and A. M. Frieze. On the connectivity of random m-orientable graphs and digraphs. *Combinatorica*, 2(4):347–359, 1982.

[44] J. Franklin and J. Lorenz. On the scaling of multidimensional matrices. *Linear Algebra and its applications*, 114:717–735, 1989.

[45] A. Frieze and T. Johansson. On random k-out subgraphs of large graphs. *Random Structures & Algorithms*, 50(2):143–157, 2017.

[46] A. M. Frieze. Maximum matchings in a class of random graphs. *Journal of Combinatorial Theory, Series B*, 40(2):196–212, 1986.

[47] A. Froger, O. Guyon, and E. Pinson. A set packing approach for scheduling passenger train drivers: the French experience. In *RailTokyo2015*, Tokyo, Japan, March 2015.

[48] M. Fürer and S. P. Kasiviswanathan. An almost linear time approximation algorithm for the permanent of a random (0-1) matrix. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 263–274. Springer, 2004.

[49] M. Fürer and S. P. Kasiviswanathan. Approximately counting perfect matchings in general graphs. In *ALENEX/ANALCO*, pages 263–272, 2005.

[50] H. N. Gabow and R. E. Tarjan. Algorithms for two bottleneck optimization problems. *Journal of Algorithms*, 9(3):411–417, 1988.

[51] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989.

[52] M. Ghaffari, K. Nowicki, and M. Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1260–1279. SIAM, 2020.

[53] A. Globerson, G. Chechik, F. Pereira, and N. Tishby. Euclidean Embedding of Co-occurrence Data. *The Journal of Machine Learning Research*, 8:2265–2295, 2007.

[54] A. Goel, M. Kapralov, and S. Khanna. Perfect matchings in $\mathcal{O}(n \log n)$ time in regular bipartite graphs. *SIAM Journal on Computing*, 42(3):1392–1404, 2013.

[55] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[56] G. Gottlob and G. Greco. Decomposing combinatorial auctions and set packing problems. *J. ACM*, 60(4):24:1–24:39, September 2013.

[57] T. Hagerup, K. Mehlhorn, and J. I. Munro. Maintaining discrete probability distributions optimally. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *20th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 253–264, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[58] M. Halldórsson. Approximating discrete collections via local improvements. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, pages 160–169, USA, 1995. Society for Industrial and Applied Mathematics.

[59] E. Hazan, S. Safra, and O. Schwartz. On the complexity of approximating k-dimensional matching. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 83–97. Springer, 2003.

[60] E. Hazan, S. Safra, and O. Schwartz. On the complexity of approximating k-set packing. *Computational Complexity*, 15(1):20–39, 2006.

[61] J. Holm, V. King, M. Thorup, O. Zamir, and U. Zwick. Random k-out subgraph leaves only o (n/k) inter-component edges. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 896–909. IEEE, 2019.

[62] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[63] Y. Huo, H. Liang, S. Liu, and F. Bai. Computing monomer-dimer systems through matrix permanent. *Physical Review E*, 77(1):016706, 2008.

[64] C. A. J. Hurkens and A. 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.

[65] M. Idel. A review of matrix scaling and Sinkhorn's normal form for matrices and positive maps. *arXiv preprint arXiv:1609.06349*, 2016.

[66] M. Jerrum, A. Sinclair, and E. Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *J. ACM*, 51(4):671–697, 2004.

[67] B. Kalantari and L. Khachiyan. On the complexity of nonnegative-matrix scaling. *Linear Algebra Appl.*, 240:87–103, 1996.

[68] M. Karoński, E. Overman, and B. Pittel. On a perfect matching in a random bipartite digraph with average out-degree below two. *arXiv preprint arXiv:1903.05764*, 2019.

[69] M. Karoński and B. Pittel. Existence of a perfect matching in a random $(1 + e^{-1})$–out bipartite graph. *J. Comb. Theory B*, 88(1):1–16, 2003.

[70] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[71] R. M. Karp, A. H. G. Rinnooy Kan, and R. V. Vohra. Average case analysis of a heuristic for the assignment problem. *Mathematics of Operations Research*, 19(3):513–522, 1994.

[72] R. M. Karp and M. Sipser. Maximum matching in sparse random graphs. In *FOCS'81*, pages 364–375, Nashville, TN, USA, 1981.

[73] R. M. Karp, U. V. Vazirani, and V. V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, pages 352–358, New York, NY, USA, 1990. ACM.

[74] P. W. Kasteleyn. The statistics of dimers on a lattice. *Physica*, 27:1209–1225, 1961.

[75] K. Kaya. Parallel algorithms for computing sparse matrix permanents. *Turkish Journal of Electrical Engineering & Computer Sciences*, 27(6):4284–4297, 2019.

[76] K. Kaya, J. Langguth, F. Manne, and B. Uçar. Push-relabel based algorithms for the maximum transversal problem. *Computers & Operations Research*, 40(5):1266–1275, 2013.

[77] O. Kaya and B. 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.

[78] P. A. Knight. The Sinkhorn–Knopp algorithm: Convergence and applications. *SIAM J. Matrix Anal. A.*, 30(1):261–275, 2008.

[79] P. A. Knight and D. Ruiz. A fast algorithm for matrix balancing. *IMA Journal of Numerical Analysis*, 33(3):1029–1047, 2013.

[80] P. A. Knight, D. Ruiz, and B. Uçar. A symmetry preserving algorithm for matrix scaling. *SIAM Journal on Matrix Analysis and Applications*, 35(3):931–955, 2014.

[81] T. C. Koopmans and M. Beckmann. Assignment problems and the location of economic activities. *Econometrica: journal of the Econometric Society*, pages 53–76, 1957.

[82] V. Korenwein, A. Nichterlein, P. Zschoche, and R. Niedermeier. Data reduction for maximum matching on real-world graphs: theory and experiments. *arXiv preprint arXiv:1806.09683*, 2018.

[83] J. Kulkarni, E. Lee, and M. 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.

[84] S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck. Finding Near-Optimal Independent Sets at Scale. In *Proceedings of the 16th Meeting on Algorithm Engineering and Exerpimentation (ALENEX'16)*, 2016.

[85] J. Langguth, F. Manne, and P. Sanders. Heuristic initialization for bipartite matching problems. *J. Exp. Algorithmics*, 15:1.3:1.1–1.3:1.22, 2010.

[86] L. Lovász and M. D. Plummer. *Matching Theory*, volume 367. American Mathematical Soc., 2009.

[87] J. Magun. Greedy matching algorithms, an experimental study. *Journal of Experimental Algorithmics*, 3:6, 1998.

[88] M. Marcus and R. Ree. Diagonals of doubly stochastic matrices. *The Quarterly Journal of Mathematics*, 10(1):296–302, 1959.

[89] N. McKeown. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking*, 7:188–201, 1999.

[90] G. B. Mertzios, A. Nichterlein, and R. Niedermeier. The power of linear-time data reduction for maximum matching. In *42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017)*, volume 83 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 46:1–46:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[91] S. Micali and V. V. Vazirani. An $\mathcal{O}(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *FOCS'80*, pages 17–27, 1980.

[92] M. Mitzenmacher and E. Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[93] R. H. Möhring and M. Müller-Hannemann. Cardinality matching: Heuristic search for augmenting paths. Technical report, Technische Universität Berlin, 1995.

[94] U. Naumann and O. Schenk. *Combinatorial scientific computing*. CRC Press, 2012.

[95] A. Nijenhuis and H. S. Wilf. *Combinatorial algorithms: for computers and calculators*. Academic Press, 1978.

[96] M. Poloczek and M. 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.

[97] A. Pothen and C.-J. Fan. Computing the block triangular form of a sparse matrix. *ACM T. Math. Software*, 16:303–324, 1990.

[98] A. Pothen, S. M. Ferdous, and F. Manne. Approximation algorithms in combinatorial scientific computing. *Acta Numerica*, 28:541–633, 2019.

[99] L. E. Rasmussen. Approximating the permanent: A simple approach. *Random Struct. Algor.*, 5(2):349–361, 1994.

[100] J. Shetty and J. Adibi. The enron email dataset database schema and brief statistical report. *Information sciences institute technical report, University of Southern California*, 4, 2004.

[101] R. Sinkhorn and P. Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific J. Math.*, 21:343–348, 1967.

[102] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. FROSTT: The formidable repository of open sparse tensors and tools. http://frostt.io/, 2017.

[103] H. N. V. Temperley and M. E. Fisher. Dimer problem in statistical mechanics-an exact result. *The Philosophical Magazine: A Journal of Theoretical Experimental and Applied Physics*, 6(68):1061–1063, 1961.

[104] G. Tinhofer. A probabilistic analysis of some greedy cardinality matching algorithms. *Annals of Operations Research*, 1(3):239–254, 1984.

[105] B. Uçar. *Partitioning, matching, and ordering: Combinatorial scientific computing with matrices and tensors*. 2019.

[106] L. G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8(2):189–201, 1979.

[107] D. Walkup. Matchings in random regular bipartite digraphs. *Discrete Mathematics*, 31(1):59–64, 1980.

List of publications

Articles in international refereed journals

[J1] F. Dufossé, K. Kaya, I. Panagiotas, and B. Uçar. Further notes on Birkhoff–von Neumann decomposition of doubly stochastic matrices. *Linear Algebra and its Applications*, 554:68–78, 2018.

[J2] F. Dufossé, K. Kaya, I. Panagiotas, and B. Uçar. Scaling matrices and counting perfect matchings in graphs. *Discrete Applied Mathematics*, to appear.

Articles in international refereed conferences

[C1] F. Dufossé, K. Kaya, I. Panagiotas, and B. Uçar. Effective heuristics for matchings in hypergraphs. In *SEA^2, International Symposium on Experimental Algorithms*, pages 248–264, Kalamata, Greece, June 2019. Springer.

[C2] K. Kaya, J. Langguth, I. Panagiotas, and B. Uçar. Karp–Sipser based kernels for bipartite graph matching. In *20th SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 134–145, Salt Lake City, Utah, US, January 2020.

[C3] I. Panagiotas and B. Uçar. In F. Grandoni, G. Herman, and P. Sanders, editors, *28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 76:1–76:23, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Articles in international refereed workshops

[W1] F. Dufossé, K. Kaya, I. Panagiotas, and B. 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.