UNIVERSITY OF CALIFORNIA,
IRVINE

Graph Construction using the dK-Series Framework

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Networked Systems

by

Balint Tillman

Dissertation Committee:
Professor and Chancellor's Fellow Athina Markopoulou, Chair
Chancellor's Professor David Eppstein
Professor Carter T. Butts

2019

DEDICATION

To my friends and family
who were there to support me through this work
and helped me to overcome obstacles along the way.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

This work would have not been possible with the guidance and support of my advisor, Prof. Athina Markopoulou. I am grateful for her invitation to work on this problem, which led me to explore and push the boundaries of what is possible within the *d*K-series framework. I would also like to thank Prof. Carter T. Butts and Prof. David Eppstein for both serving on my defense committee and their collaboration and support throughout the past years. Thank you to my collaborators: Dr. Minas Gjoka for introducing me to the first problems discussed in this dissertation and Dr. William E. Devanny for our collaboration on the NP-Hardness results. Thanks to past and current students from my lab for their support and helpful insights: Dr. Blerim Cici, Dr. Luca Baldesi, Dr. Omer Nebil Yaveroglu, Dr. Anastasia Shuba, Emmanouil Alimpertis, Evita Bakopoulou, Milad Asgari, Janus Varmarken, Hieu Van Le. Thank you to Prof. Aparna Chandramowlishwaran and Prof. Ardalan Amiri Sani for their service on my candidacy committee. Thank you is also in order for my mentors and coworkers at Google: Sam Aldrin, Dr. Sriram Natarajan, Alexandru Mosoi, Dr. Jonathan Halcrow and Dr. Bryan Perozzi.

I have greatly appreciated all the funding support from the University of California, Irvine through the Networked Systems Fellowship and the Henry Samueli Fellowship; the National Science Foundation Awards 1526736 and 1028394; and Google for the summer internships, this work could not have been accomplished without these sources.

I would also like to thank IEEE to give permission to use previously published work in this dissertation. Portions of this dissertation's text are a reprint of the material as it appears in Balint Tillman, Athina Markopoulou, Minas Gjoka, and Carter T. Butts: "2K+ graph construction framework: Targeting joint degree matrix and beyond", IEEE/ACM Transactions on Networking, 27(2):591-606, April 2019. Athina Markopoulou listed in this publication directed and supervised research which forms the basis for the dissertation.

Last but not least, I would like to thank all my friends near and far for their presence and support throughout this journey. I would like thank my family for their support and patience that allowed me to focus on this work for the past years.

CURRICULUM VITAE

Balint Tillman

EDUCATION

Doctor of Philosophy in Networked Systems — **2019**
University of California, Irvine — *Irvine, California, USA*

Master of Science in Software Development and Technology — **2014**
IT University of Copenhagen — *Copenhagen, Denmark*

Bachelor of Science in Business Information Technology — **2012**
Corvinus University of Budapest — *Budapest, Hungary*

RESEARCH EXPERIENCE

Graduate Research Assistant — **2014–2019**
University of California, Irvine — *Irvine, California, USA*

Visiting Researcher — **2014**
University of California, Irvine — *Irvine, California, USA*

Visiting Researcher — **2014**
University of Amsterdam — *Amsterdam, Netherlands*

TEACHING EXPERIENCE

Teaching Assistant for EECS215 — **2015, 2018**
University of California, Irvine — *Irvine, California, USA*

Teaching Assistant for SAD1 and SGDS — **2013**
IT University of Copenhagen — *Copenhagen, Denmark*

ABSTRACT OF THE DISSERTATION

Graph Construction using the dK-Series Framework

By

Balint Tillman

Doctor of Philosophy in Networked Systems

University of California, Irvine, 2019

Professor and Chancellor's Fellow Athina Markopoulou, Chair

It is often desirable to generate random graphs that exhibit certain prescribed properties, for example, for simulation or anonymization of real-world graphs. In this dissertation, we adopt and build on the dK-series modeling framework and we make the following three sets of contributions. First, we focus on undirected graphs, and we provide a flexible approach for generating simple undirected graphs with the exact target joint degree matrix (JDM), which we refer to as 2K graphs, in linear time in the number of edges. Our 2K construction algorithm imposes minimal constraints on the graph structure, which allows us to target additional graph properties, namely node attributes (2K+A), clustering (both average clustering, 2.25K, and degree-dependent clustering, 2.5K) and number of connected components (2K+CC). We show that realizability of exact 2.25K, 2.5K and in general dK-series for any $d \geq 3$ is NP-Complete. Second, we also define the problem of directed and directed acyclic 2K graph construction (D2K, DAG2K), we provide necessary and sufficient conditions for realizability, we develop efficient construction algorithms for D2K and solve DAG2K in the special case of level graphs (D2K+L). We evaluate our approach by creating synthetic graphs that target real-world graphs both undirected (such as Facebook) and directed (such as Twitter) and we show that it brings significant benefits, in terms of accuracy and running time, compared to state-of-the-art approaches.

Third, we propose a new approach for graph construction with a prescribed local neighborhood structure. To capture that structure, we define the Neighborhood Partition Matrix (NPM): given a node partition, NPM specifies the number of edges from each node to each part of the partition. Our goal is to construct simple graphs that exhibit a given NPM, if that is realizable. NPM can also be thought of as graph embedding, where each dimension corresponds to a part in the partition. This key observation allows us to create graph realizations from these embeddings with guarantees such as degree sequences, degree correlations, etc. The main strength of the NPM approach is its generality: (i) it bridges the gap between classic graph realization (for degree sequences and other structural properties) and graph embeddings; (ii) it allows arbitrary node partitions, thus includes prior models as special cases. We describe the main approach for undirected graphs and we extend it to directed graphs and graphs with non-chords. We evaluate strategies for NPM based on different node partitions and we compare against baselines w.r.t. targeted graph properties and graph embedding tasks, namely, graph reconstruction, link prediction and node classification.

Chapter 1

Introduction

1.1 Motivation

It is often desirable to generate synthetic graphs that resemble real-world networks with regards to certain properties of interest. For example, researchers often want to simulate a process on a realistic network topology but they may not have access to a real-world network; or they may want to generate several different realizations of graphs of interest. In this dissertation, we target both directed and undirected graphs including, but not limited to, online social networks.

There is a large body of work, in classic literature [23],[42],[41],[64], as well as more recently [52],[57], [24], on generating realizations of undirected graphs that exhibit (exactly) target structural properties such as a degree sequence or a joint degree matrix. In this dissertation, we adopt the dK-series framework [52],[57], which describes graphs in terms of a series of frequencies of degree-labeled induced subgraphs of increasing size, thus providing an elegant way to trade off accuracy (in terms of graph properties) vs. complexity (of the algorithms generating graph realizations with those properties). Construction of 1K-graphs (*i.e.* graphs

with a target degree sequence) is well understood: efficient algorithms and realizability conditions are known since Havel-Hakimi [42],[41]. Construction of 2K-graphs (graphs with a target joint degree matrix) has been studied in parallel by several researchers, namely Amanatidis et al. [73], Czabarka et al. [16], and our group [34]. For $d > 2$, which is necessary for capturing the clustering exhibited in social networks, we recently proved that the problem is NP-hard [19] and we developed efficient heuristics [34]. In contrast, construction of directed graphs was not as well developed: results were known for construction of graphs with a target directed degree sequence [30], [29], but the notion of directed degree correlation, or directed dK-series for $d \geq 2$, has not been previously defined.

In this dissertation, we present a general algorithmic framework that allows us to construct synthetic graphs with an exact target JDM (which we refer to as "2K" graphs) and potentially additional properties (which we refer to as "2K+" graphs). The core of our 2K+ framework is an algorithm that can provably create synthetic undirected graphs with the exact target JDM, and it does so efficiently (*i.e.* in linear time in the number of edges), while imposing minimal constraints on the graph structure (*i.e.* it can potentially create any 2K realization). We exploit the latter feature to impose additional properties during construction, namely node attributes (2K+A), clustering (both average clustering, 2.25K, and degree-dependent clustering, 2.5K) and number of connected components (2K+CC).

We also extend the 2K framework, for the first time, to directed graphs. We define two notions of degree correlation in directed 2K graphs: directed 2K (D2K) and its special case D2.1K. D2K includes the notion of directed degree sequence (DDS) and maps directed graphs to bipartite undirected graphs to also express degree-correlation via a joint degree-attribute matrix (JDAM) for the bipartite graph. This problem definition lends itself naturally to techniques we previously developed for undirected 2K [34], which we exploit to develop (i) necessary and sufficient realizability conditions and (ii) an efficient algorithm that constructs graph realizations with the exact target D2K. In addition, we also extend this approach to

acyclic graphs and solve a special case of level graphs (D2K+L).

Finally, we present a new graph construction problem which we call the Neighborhood Partition Matrix (NPM) problem. The input to the problem is NPM with corresponding partition P, which captures the local neighborhoods in the following sense: $NPM[i, j]$ captures the number of edges form node i to nodes in part j of P. This is a generalization of Degree Spectra Matrix (DSM) [9] and Neighborhood Degree List (NDL) [8] from partitioning nodes by degree (which only captures specific structural properties) to arbitrary partitions. This generalization allows us to utilize NPM as a graph embedding problem with arbitrary dimensions. This bridges the gap between two previously disconnected literatures: the classical graphical construction and graph embeddings, allowing us to leverage existing techniques and applications of both. To the best of our knowledge, we are the first to propose such graph construction-based models as graph embeddings. More specifically, our method is flexible with regards of node partitioning compared DSM or NDL, while the fundamental building blocks of these methods are very similar. This means that many related works that DSM and NDL builds on can be also used in our context. We highlight the relation to other graph construction problems and we discuss extensions of NPM to capture other properties, in addition to NPM, clustering coefficients, directed graph construction.

1.2 Problem Statement

When constructing a synthetic graph that resembles a real graph G, we have to specify several aspects of the problem.

First, we have to choose the properties of G that should be preserved: this is in itself a challenging research question. We adopt the systematic framework of *dK-series* from Mahadevan *et al.* [52], which was introduced to characterize the properties of a graph

using a series of probability distributions specifying all degree correlations within d-sized, simple, and connected subgraphs of a given graph G. In this framework, higher values of d capture progressively more properties of G at the cost of more complex representation of the probability distribution. The dK-series exhibit two desired properties: inclusion (a dK distribution includes all properties defined by any d'K distribution, $\forall d' < d$) and convergence (nK, where $n = |V|$ specifies the entire graph, within isomorphism). [1]

Second, we have to define in what sense the synthetic graph should resemble the original one. In this dissertation, we produce *simple* graphs that exhibit the target properties *exactly*. This is different from the stochastic approach presented by [22] (target properties are achieved in expectation) or the configuration model in [3] (graphs could be multigraphs as well).

Depending on how probabilistic construction is performed, its realizations may be associated with JDMs that are far from the target, which may or may not be desirable in practice. While our focus in this dissertation is on exact construction, we note that probabilistic and deterministic construction are complementary approaches in a broader graph construction toolkit and can be used together. Exact construction can facilitate probabilistic construction by, for example, first simulating JDMs from a target distribution and then construction graphs satisfying those simulated JDMs (if necessary, filtering out unfeasible JDMs as a form of rejection sampling). Many approaches to probabilistic simulation of complex network distributions (*e.g.* those based on Markov chain Monte Carlo or related methods) are fairly expensive, and exact construction may be faster (provably on the order of the number of edges) for large graphs.

Third, we have to specify what realizations with the target properties can be achieved and how: at least one (if such exists), all possible realizations (and the corresponding sampling

[1] A study of how well dK-series match real-world graphs was conducted by Orsini *et al.* in [57]. Six real-world undirected graphs were considered and compared to synthetic graphs produced by dK-series in terms of a range of graph properties (from local to global, targeted and non-targeted). The paper [57] demonstrated the convergence of dK-series for these graphs and properties, for $d \leq 2.5$, in the overwhelming majority of the cases.

method), a subset of realizations, etc.

A **dK-construction problem** takes as input the target properties[2] (*i.e.* the dK-series and potentially additional properties), and addresses the three following subproblems.

- **Realizability:** Provide necessary and sufficient conditions such that there exist simple graphs with these target properties.
- **Construction:** Design an algorithm that generates at least one such graph realization.
- **Space of realizations:** Characterize the space of all graph realizations with these target properties and provide ways to sample from them.

In the next subsections, we discuss in detail the dK-series framework [52] and summarize prior work.

1.3 Our Work in Perspective

1.3.1 Prior Work on Undirected Graph Construction

Consider an undirected graph $G = (V, E)$, with $n = |V|$ nodes and $m = |E|$ edges. Let $\deg(v)$ be the degree of node v. Let V_k be the set of nodes that have degree k, also referred to as degree group k.

0K Construction. 0K describes graphs with a prescribed number of nodes and edges. This notion corresponds to simple Erdős-Rényi (ER) graphs with fixed number of edges.

[2] We use $\odot$ to denote the target properties, that the constructed graph should have; absence of $\odot$ denotes the actual values for the constructed, or partially constructed, graph.

1K Construction. In an undirected graph G, a node v has degree $deg(v)$, $D_k = |V_k|$ is the number of nodes of degree $k, k = 1, ..., d_{max}$, where d_{max} is the maximum degree in the graph. The degree sequence is simply:

$$DS = \{deg(v_1), deg(v_2), ..., deg(v_{|V|})\} \tag{1.1}$$

In the dK-series terminology, the degree sequence specifies 1K. Degree sequences have been studied since the 1950s, thus we only focus on the most relevant results. The realizability conditions for degree sequences were given by the Erdős-Gallai theorem [23], and first algorithm to produce a single realization by Havel-Hakimi [42],[41]. The space of simple graph realizations of 1K distributions is connected over double edge swaps preserving degrees [64]. More recently, importance sampling algorithms were proposed in Blitzstein *et al.* [12] and Genio *et al.* [18].

2K Construction. A Joint Degree Matrix (JDM) is given by the number of edges between nodes of degree k and l [3]:

$$JDM(k,l) = \sum_{v \in V_k} \sum_{w \in V_l} 1_{\{(v,w) \in E\}} \tag{1.2}$$

Degree assortativity is a scalar that is often used to summarize JDM. A given 2K (JDM) also fixes 1K (the degree vector D_k):

$$D_k = |V_k| = \frac{1}{k} \sum_{l=1}^{d_{max}} \text{JDM}(k,l) \tag{1.3}$$

as well as the number of edges $m = |E|$, and the number of nodes $n = |V|$ in the graph, thus 0K.

[3] In case of $k = l$, this notation returns twice the number of edges within degree group k, resulting in minor differences in notation from related work.

Realizability conditions for undirected 2K were provided by Amanatidis *et al.* [73][4]. Algorithms for generating realizations of a target JDM were provided in Czabarka *et al.* [16], Gjoka *et al.* [34] and Stanton and Pinar [63]. The algorithms presented in [4] and [63] are designed to produce restricted realizations that exhibit the Balanced Degree Invariant (BDI) property, which evenly spreads edges between degree groups. In [16] and [34], the algorithms have non-zero probability to produce any realization of a 2K distribution. Bassler *et al.* [9] introduced an importance sampling algorithm. We provide an overview of the relation of these algorithms in Section 2.2.3.

W.r.t. sampling, the space of 2K-graphs with a realizable JDM was shown to be closed under JDM-preserving double edge swaps [16, 4], [4] and can be used to generate approximate probability samples for 2K. However, fast mixing has not been proved for 1K or 2K, apart from special classes of realizations [28] [27]. We further discuss these types of swaps in Section 2.2 in relation to graph properties like number of connected components and clustering.

dK, $d > 2$ Construction. While algorithms of known time complexity exist for $d \leq 2$, Monte Carlo Markov Chain (MCMC) approaches are typically used for $d > 2$. Several attempts were made to find polynomial time algorithms to produce 3K graphs [52] or 2K realizations with prescribed (degree-dependent) clustering coefficient[33],[34] and [57], but a member of our team and collaborators recently proved that checking realizability of these inputs is NP-Complete [19].

Annotated graph construction was proposed by Dimitropoulos et al. in [21] that considered degree correlations, however the proposed construction method will generate graphs with self-loops or multi-edges initially. An additional step removes these extra edges to make the graph simple and finally the largest connected component of the graph is returned.

[4] In [4] an induction proof is provided that shows swaps to reduce the symmetric difference between any two realizations until the difference disappears, a simple example is shown in Fig. 2.4. In [16] a different approach is shown where a swap sequence is given to reach a BDI realization and later to reduce the problem to a union of unipartite (1K) and bipartite degree sequence problems which have known solutions.

Partition Adjacency Matrix problem (PAM) is another relaxation of JDM construction by Erdős *et al.* [24], where the number of edges are defined over a partition of nodes (not necessarily forming degree groups) and each part of the partition is associated with a degree sequence. In [24], a construction algorithm was shown to solve the problem for any bipartition and other special cases. However, there is no general solution available for PAM and it is believed that even the realizability question is NP-Complete. The Bipartite Partition Adjacency Matrix (BPAM) was recently proposed by Czabarka et. al. [17]. The authors consider a PAM problem with a bipartition of parts such that every realization of a BPAM would be bipartite, we use a similar idea in our D2K construction. BPAM with any set of non-chords can be solved with high probability using a high-order polynomial time algorithm.

Last but not least,we highlight the two most relevant and equivalent problems to our proposed method in Chapter 4. The *Degree Spectra Matrix (DSM)* [9] and the *Neighborhood Degree List (NDL)* [8] construction problems.

Definition 1.1. *Given an undirected graph $G(V, E)$, DSM is a $|V| \times d_{max}$ matrix, where d_{max} is the largest degree in G, s.t. $DSM[i, j]$ is the number of edges from node i to nodes with degree j.*

Definition 1.2. *Given an undirected graph $G(V, E)$, NDL is a set of list, where each node i in V with degree k has a list of length k, where items are the degrees of i's neighbors.*

Both problems describe essentially the same input in different formats, and both decompose into uni- and bipartite degree sequence problems. In [9], DSM is used to sample JDM realizations, while the authors of [8] show that the space of realizations for NDL/DSM is connected over double-edge swaps. NDL/DSM are special cases of NPM where the partition of nodes are by degrees. In [8], the authors provide a sufficient characterization for NDL with a unique realization.

1.3.2 Prior Work on Directed Graph Construction

We extend the taxonomy of dK-series, from undirected graphs to also describe properties of directed graphs.

Directed 0K. There is a simple extension of ER graphs to generate directed graphs as well, which we use in our evaluation. In addition, we consider the UMAN model [43], which captures the number of mutual, asymmetric, and null dyads in a graph: UMAN can be thought of as 0K with fixed numbers of mutual and unreciprocated edges.

Directed 1K. In a directed graph, a node v has both in and out degree (d_v^{in}, d_v^{out}) and the *directed degree sequence* can be expressed as follows.

$$DDS = \{(d_v^{in}, d_v^{out}), v \in V\} \tag{1.4}$$

It is well known from Gale's work [30], that any directed graph can be mapped 1-1 to an undirected bipartite graph, where each node v of the directed graph is split in two nodes v_{in} and v_{out}, and the undirected edges across the two (in and out) partitions of the bipartite graph correspond to the directed edges in the directed graph. A self loop (v, v) in the directed graph corresponds to a "non-chord" (v_{in}, v_{out}) in the bipartite graph. We further discuss this transformation and provide examples in Section 3.2.

Construction algorithms are known for a bipartite degree sequence with [30], or without non-chords, and therefore for the corresponding directed graphs with or without [29] self-loops, respectively. More recently, an importance sampling algorithm was provided for D1K in [48]. In general, Tutte's gadget [70] can also solve DS, BDS, DDS problems with any set of non-chords as pointed out by Erdős et. al [26]. However, this approach does not scale well because of its $O(|V|^4)$ complexity.

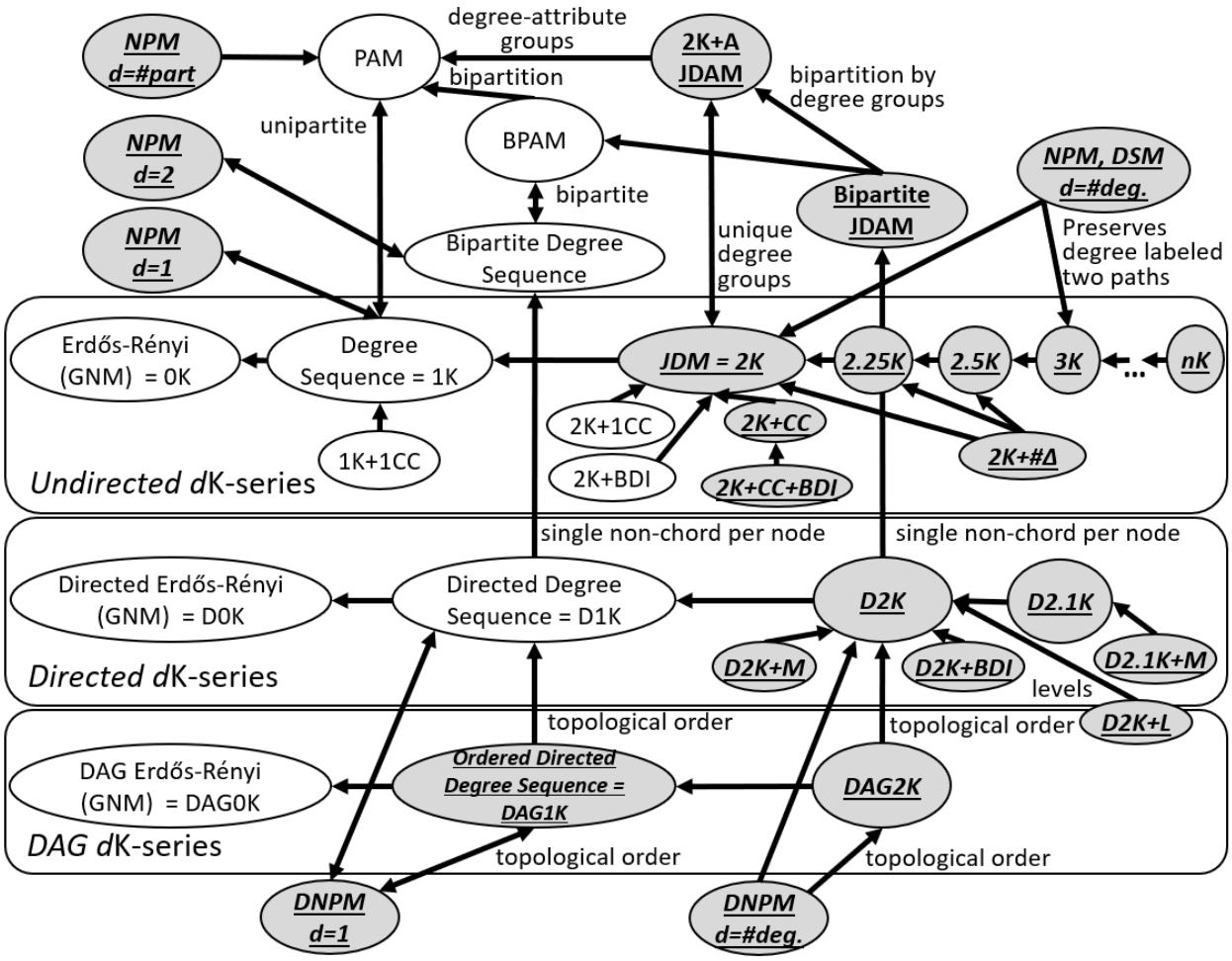


Figure 1.1: Overview of graph construction problems, and the relation between them. An arrow from P1 to P2 means that problem P2 fixes more properties and bidirectional arrows show equivalence between problems under certain conditions. The contributions made in this dissertation are highlighted in gray.

1.3.3 Dissertation Contributions: The 2K+ Framework

The proposed 2K+ framework not only constructs graphs with an exact target JDM but also targets additional properties. This dissertation unifies and extends a number of our previously disconnected results, which have partially appeared in conferences and workshops, on construction of undirected (2K [33][35], 2.25K [34], 2.5K [33], 2K+CC [66]) as well as directed graphs (D2K, D2.1K [67]). Figure 1.1 depicts an overview of the problem space including past work and our contributions.

2K Construction. JDM realizations have been studied independently and in parallel by several groups. The problem was spearheaded by Amanatidis *et al.* first in an unpublished manuscript in 2008 (on the realizability conditions), and in a more recent arxiv in 2015 [4] (on construction algorithms for balanced and single connected component realizations). It was followed by Stanton and Pinar in 2012 [63] also targeting BDI realizations and a proof for connected space that was found to have flaws. Around the same time, construction algorithms that could also produce non-balanced realizations were developed by two other groups: Czabarka *et al.* [16] in 2015, and by Gjoka *et al.* in 2013 [33] and 2015 [34] (ours). The common idea behind all construction algorithms is to add one edge at the time so as to not exceed node degrees or JDM; algorithms vary on if/how they violate any of these two properties and on when/how to correct it using local rewiring; which is always possible for realizable JDMs. In addition to independently developing one of the 2K algorithms in [33, 34], our contribution in this part lies in the particular order in which we add and rewire edges, which is necessary for being able to target additional properties and enable a framework beyond just 2K.

Use cases: The 2K distribution captures properties such as differential mixing by degree, which can be important for modeling phenomena such as diffusion. In particular, in a degree-conditioned random graph, high-degree nodes are proportionally more likely to be adjacent to each other than to low-degree nodes; this produces a core in the network, and high connectivity among hubs (particularly where the degree distribution is highly skewed), leading to rapid hub-based diffusion. In real networks, however, one may see other mixing patterns involving, for example, higher or lower levels of assortative degree mixing, or entirely different patterns (*e.g.* a tendency for degree-1 nodes to mix with each other, producing large numbers of isolated dyads). Matching the 2K distribution ensures that these properties are accurately represented. It is interesting to note that 2K can match these properties. while 1K cannot, while having the same linear complexity $O(|E|)$.

Clustering: 2.25K and 2.5K. 2K (JDM) in itself does not capture clustering, which is an essential property of several real-world graphs such as online social networks. 3K captures a very strong notion of clustering, whose construction we recently showed to be NP-hard [19]. Our main motivation behind the 2K+ construction work was to efficiently generate graphs with a target JDM *and* some notion of clustering and we targeted two such notions: average clustering $\bar{c}$ and average degree-dependent clustering $\bar{c}(k)$.

In total there are four main clustering definitions we consider in this dissertation:

- The global clustering coefficient is defined as the three times the number of triangles divided by the number of two paths, that is equivalent to the following:

$$c_G = \frac{\sum_{v \in V} t_v}{\sum_{v \in V} \binom{deg(v)}{2}}, \tag{1.5}$$

  where t_v is the number of triangles touching node v.

- The local clustering coefficient captures the inter-connectivity of a node's neighborhood:

$$c_v = \frac{t_v}{\binom{deg(v)}{2}}. \tag{1.6}$$

- We define the degree-dependent clustering coefficient for degree k is defined as the average of local clustering coefficients of degree k nodes:

$$\bar{c}(k) = \frac{\sum_{v \in V_k} c_v}{|V_k|} = \frac{\sum_{v \in V_k} t_v}{|V_k| \cdot \binom{k}{2}}. \tag{1.7}$$

- The average clustering coefficient is defined as the average of local clustering coefficients

of all nodes:

$$\bar{c} = \frac{\sum_{v \in V} c_v}{|V|}. \tag{1.8}$$

In the **2.25K** problem, we develop a construction approach to target JDM and $\bar{c}$ [34]. In the **2.5K** problem, we develop a hybrid construction and MCMC approach to target JDM and $\bar{c}(k)$ [34, 33], efficiently. The heuristic nature of our approaches is justified by the hardness of 2K construction with *any* notion of clustering and the computational hardness of the dK-series for fixed $d \geq 3$ [19] as we include these results here as well.

Use cases: 2.25K and 2.5K distributions capture clustering, which is also important for diffusion: it is well known that clusters are the obstacle to information cascades over networks. Online social networks exhibit high clustering (e.g., compared to random graphs) and this is one of the main motivations for this work: producing synthetic graphs that resemble real, large online social network graphs in reasonable time. As we will see in the evaluation results, prior MCMC-based approaches targeting clustering on large online social networks do not converge in weeks, while our 2.25K and 2.5K construction converged on the order of minutes.

Number of Connected Components: 2K+CC. A single connected component (which we refer to as 1CC) can be targeted in addition to the degree sequence [64, 71], or in addition to a target JDM [4]. Our result builds on and extends Amanatidis et al. [4] to target minimum number of connected components for a given JDM, and we show that the space of those realizations is connected under JDM-preserving double-edge swaps; a one-page abstract is at [66] and the extended version and proofs are provided in this dissertation.

Node Attributes: 2K+A. In order to capture attributes in addition to structural properties, we were the first to define and target the Joint Degree-Attribute Matrix (JDAM), which captures correlation between node degrees and attributes. We show that our 2K construction

algorithms gracefully extends to JDAM construction [34]. This is useful not only for incorporating attributes into the model, but also for imposing additional structure by properly assigning the attributes in JDAM, such as bipartite JDAMs and community membership.

D2K Construction. We have applied our 2K approach to define and solve the directed 2K graph construction problem. We also show several extension (similar to 2K), such as D2.1K that captures average neighbor degree, a heuristic to target number of mutual edges (reciprocated edges in directed graphs) and balanced realizations. The observations used to show how to construct D2K balanced realizations are also used to provide an importance sampling algorithm for D2K based on work from Bassler *et al.* [9].

DAGdK-series. We describe results for directed acyclic graph construction. First we show a simple reduction to network flow problems to solve Ordered Directed Degree Sequence problem for simple graphs. Second we highlight possible approaches to solve the general DAG2K construction problem (ODDS and degree correlations) and finally, provide a special case for D2K with level assignments where every realization will be both acyclic and confirm to a level assignment similar to graph drawings.

Neighborhood Partition Matrix: NPM. We present a new graph construction problem which we call the Neighborhood Partition Matrix (NPM) problem. The input to the problem is NPM with corresponding partition P, which captures the local neighborhoods in the following sense: $NPM[i, j]$ captures the number of edges form node i to nodes in part j of P. This is a generalization of Degree Spectra Matrix (DSM) [9] and Neighborhood Degree List (NDL) [8] from partitioning nodes by degree to arbitrary partitions. We highlight the relation to other graph construction problems and we discuss extensions of NPM to capture other properties, in addition to NPM, clustering coefficients, directed graph construction.

Compared to other graph embedding methods discussed in Section 4.2, NPM has two qualitative advantages: (i) it creates graph realizations that exhibit exactly the target properties

(such as degree sequences, degree correlations, etc); (ii) it is more interpretable. NPM also has some technical differences, *i.e.* the embedding vectors have integer instead of continuous values. Compared to graph construction, the decomposition of NPM leverages efficient approaches but generalizes beyond partitioning by degree only. In our evaluation, we show that NPM outperforms baseline graph embedding methods for graph construction and graph reconstruction, for several datasets, and performs comparable to baselines on link prediction and node classification tasks. Thus, the NPM model brings qualitative improvements (flexibility and interpretability), while also achieving better or – in the worst case – similar performance w.r.t. all tasks.

Chapter 2

Undirected Graph Construction

2.1 Introduction

In this chapter, we focus of the undirected dK-series. First, we consider 2K graph construction: we define necessary and sufficient conditions for realizability and we provide a simple, efficient algorithm to construct realizations. We exploit the flexibility of our 2K algorithm and we develop heuristics to generate 2.25K graphs and speed up 2.5K targeting MCMC samplers. We also extend our method to the combination of degrees and attributes, called JDAM (2K+A). We show that it is possible to efficiently construct 2K realizations with minimum number of connected components and that the space of realizations are connected over double-edge swaps below a target number of connected components. Finally we show that the realizability problems for dK-distributions are NP-Complete for fixed $d \geq 3$.

The outline of this chapter is as follows. Section 2.2 defines and solves the basic 2K-construction problem, while Section 2.3 extends it to the 2K+ framework for 2K with additional properties (2.25K, 2.5K, 2K+A, 2K+CC). Section 2.4 provides the proofs for the computational hardness of dK-series for $d \geq 3$. Section 2.5 provides a comparison to

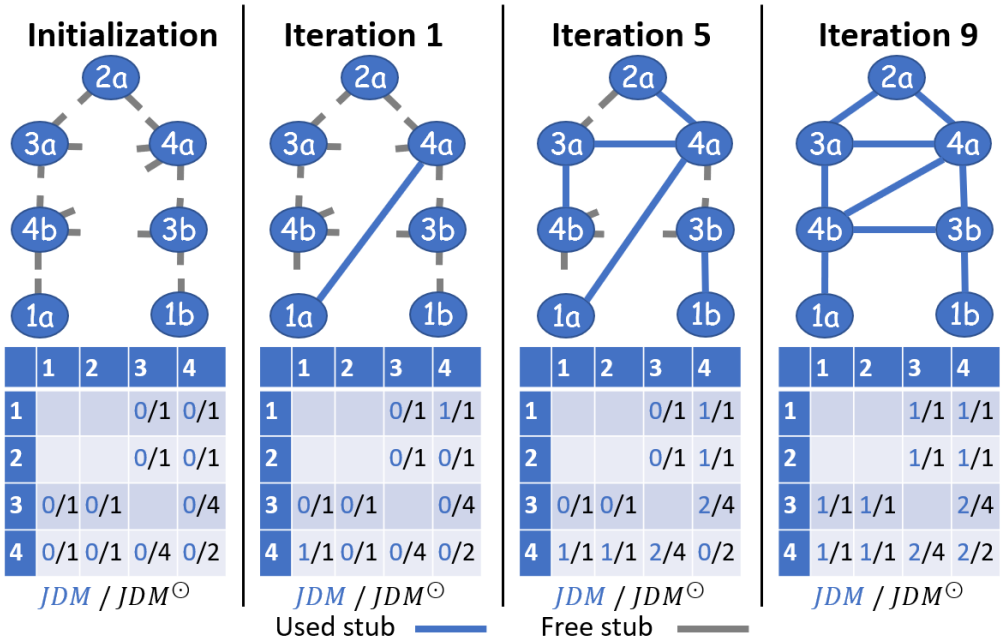


Figure 2.1: Example of running `2K_Simple`. The algorithm starts from nodes with only free stubs (left). In each iteration it creates one edge by connecting 2 free stubs and updates corresponding JDM entries until the graph is complete (right).

state-of-the-art methods when targeting several real-world undirected graphs. Section 2.6 summarizes this chapter.

2.2 2K Construction: JDM

In this section, the input is the target JDM$^{\odot}$, and the goal is to create a (at least one) simple undirected graph with N nodes that exhibits that exact JDM$^{\odot}$, if it is realizable.

2.2.1 Realizability

Not every target JDM is realizable (or "graphical"): there does not always exist at least one simple graph with this exact property. Necessary and sufficient conditions for a target JDM to be realizable, have been developed independently [4, 63, 16, 34] and are the following:

I $\forall k,\ JDM(k,k) \leq |V_k| \cdot (|V_k| - 1)$

II $\forall k, l, k \neq l,\ JDM(k,l) \leq |V_k| \cdot |V_l|$

III $\forall k : |V_k| = \sum_l \frac{JDM(k,l)}{k}$, and it is an integer.

These conditions are necessary and describe intuitive conditions for inputs to be realizable. Violating the first condition would necessarily result in a multi-graph or graphs with self-loops, since it describes the number of edges contained in a complete graph for a degree group. Similarly the second condition describes a complete bipartite graph between a pair of degree groups. The third condition ensures that size of degree groups are integers and gives the number of nodes with certain degree. Sufficiency of these conditions are shown by a constructive proof of our algorithm.

2.2.2 Algorithm for 2K Construction

`2K_Simple` receives a target JDM$^{\odot}$ as input and creates a simple undirected graph with JDM$^{\odot}$. It is summarized next and is illustrated in the example of Fig. 2.1.

The initialization phase is depicted in the leftmost column of Fig. 2.1. We create $|V| = n$ nodes, labeled by their degree; note that $|V|$ and $D_k^{\odot}$ can be found from JDM$^{\odot}(k,l)$. Following the configuration model approach, we assign k free stubs to every node $v \in V_k$ according to their degree. Stubs are the "half" edges shown in the top-left part of Fig. 2.1, originally free, *i.e.* not connected to any other nodes. We also initialize all entries of $JDM(k,l)$ to zero.

Then the algorithm proceeds in iterations by adding one edge at a time until JDM matches $JDM^{\odot}$. More specifically, we pick two nodes v and w from degree groups V_k and V_l respectively, s.t. $JDM(k,l)$ has not reached its target yet (*i.e.* $JDM(k,l) < JDM^{\odot}(k,l)$ in line 2

Algorithm 2.1 `2K_Simple`

Input: $JDM^{\odot}$
Initialize:
a: Create $|V|$ nodes; each $v \in V$ has $deg(v)$ free stubs
b: Set $JDM(k,l) = 0$ for every $(k,l) \in JDM^{\odot}$
Add Edges:
1: **for** $(k,l) \in JDM^{\odot}(k,l)$
2: **while** $JDM(k,l) < JDM^{\odot}(k,l)$
3: Pick any nodes $v \in V_k$ and $w \in V_l$
s.t. (v,w) is not an existing edge
4: **if** v does not have free stubs:
5: v': node in V_k with free stubs
6: NeighborSwitch(v,v')
7: **if** w does not have free stubs:
8: w': node in V_l with free stubs
9: NeighborSwitch(w,w')
10: add edge between (v, w)
11: $JDM(k,l)$++ ; $JDM(l,k)$++
Output: simple undirected graph with $JDM = JDM^{\odot}$

of Algorithm 2.1 and (v,w) is not an edge. Then the algorithm connects two of their stubs to create an edge. Furthermore, the algorithm should be able to add an edge even if one or both nodes do not have free stubs. In that case, Lemma 2.3 guarantees that we will always be able to perform edge rewiring, which we refer to as *NeighborSwitch*, so as to free stubs for v and/or w.

More specifically, a NeighborSwitch for a given node w frees a stub for node w and preserves the current JDM without creating multi-edges or self-loops. It does so, if also given a node w' with the same degree as w and a free stub. First, we find a neighbor t of w such that t is not a neighbor of w', then removing edge (w,t) and adding edge (w',t). The following pseudocode summarizes a NeighborSwitch, which is also illustrated in Fig. 2.2.

Algorithm 2.2 NeighborSwitch(node w,w')

1: find t: neighbor of w and not neighbor of w'
2: remove edge (w,t)
3: add edge (w',t)

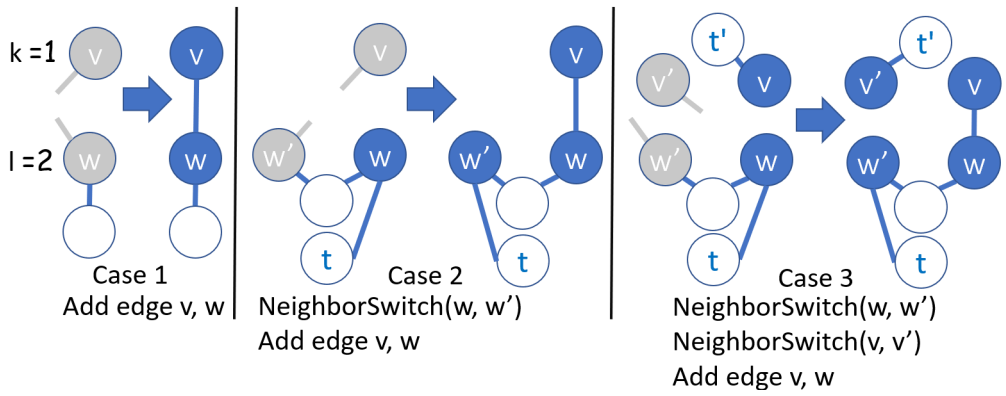


Figure 2.2: All possible cases of adding an edge (v, w) between node v (of degree k) and node w (of degree l). Blue color nodes are without free stubs and grey color indicates nodes that have remaining free stubs.

Correctness. `2K_Simple` terminates and constructs a simple undirected graph with the exact $JDM^{\odot}$.

Proof. In each iteration, `2K_Simple` adds exactly one edge making sure that JDM values never exceed target $JDM^{\odot}$ value: *i.e.* $JDM(k,l) \leq JDM^{\odot}(k,l)$. Starting from an empty graph, the algorithm adds $|E|$ edges and then terminates with $JDM = JDM^{\odot}$. Lemma 2.1 shows that the algorithm will not get stuck, *i.e.* if we have not reached the target, it is possible to find $v \in V_k, w \in V_l$ nodes where an edge can be added. There are three cases depending on whether v, w have free stubs or not:

Case 1. Add a new edge between v, w nodes w/free stubs, no local rewiring needed.

Case 2. Add a new edge between a node v w/out free stubs and a node w w/free stubs. Lemma 2.2 shows that there is $v' \in V_k$ w/free stubs and Lemma 2.3 shows that Neighbor-Switch can be applied for v, v' and it will free up a stub for v. Then (v, w) edge can be added without further rewiring.

We have to consider whether NeighborSwitch operation can add (v, w) edge if $k = l$ and $w = v'$ such that it remains possible to add it after the switch. Node t used during the switch is different from w, thus the edge added during the switch is different from (v, w).

Case 3. Add a new edge between v, w nodes w/out free stubs. Similar to Case 2. application of Lemma 2.2 gives $v' \in V_k$ and $w' \in V_l$ w/free stubs for v, w respectively. NeighborSwitches can be then applied to v, v' and w, w'. The resulting free stubs for v, w can be used to add (v, w).

Subsequent applications of NeighborSwitches will not add (v, w) even if $k = l$, because the first switch clearly uses $v' \neq w$ and the second can be handled as in Case 2. □

Lemma 2.1. *If $JDM(k,l) < JDM^{\odot}(k,l)$, then an edge can be added between V_k and V_l .*

Proof. This follows from realizability conditions [I] and [II]. Let us assume that it is not possible to add a new edge between nodes in V_k and V_l. This implies that nodes in V_k and V_l and current edges build a complete bipartite graph (or complete graph if $k = l$):

$$JDM(k,l) = JDM_{max}(k,l) = \begin{cases} |V_k| \cdot |V_l|, & \text{if } k \neq l \\ |V_k| \cdot (|V_k| - 1), & \text{if } k = l \end{cases} \tag{2.1}$$

Since $JDM^{\odot}$ for a realizable $JDM^{\odot}(k,l) \leq JDM_{max}(k,l)$, $\forall (k,l)$, which leads to a contradiction with $JDM(k,l) < JDM^{\odot}(k,l)$. □

Lemma 2.2. *If $JDM(k,l) < JDM^{\odot}(k,l)$, there is at least one node $x_k \in V_k$ with free stubs of degree k and one node $x_l \in V_l$ with free stubs.*

Proof . Let us assume that there is no node of degree k with free stubs. This means that every node $x \in V_k$ has k connected stubs and zero free stubs. This happens in two cases:

- $\forall m,\ JDM(k,m) = JDM^{\odot}(k,m)$, which contradicts $JDM(k,l) < JDM^{\odot}(k,l)$.
- $\exists m : JDM(k,m) > JDM^{\odot}(k,m)$, which contradicts the algorithm's invariant that $\forall (k,l), JDM(k,l) \leq JDM^{\odot}(k,l)$ (lines 2 and 9 of `2K_Simple` algorithm).

This is a contradiction. So does assuming that no x_l of degree l, $k \neq l$ has free stubs. □

Lemma 2.3. *NeighborSwitch is possible to execute and it is JDM preserving if* $w, w' \in V_k$ *and* $deg(w') < deg(w)$.

Proof. Since $deg(w') < deg(w)$, there exists a node t ($t \neq w', t \in V_l$, where k could be equal to l), which is a neighbor of w but not a neighbor of w'. Therefore, it is possible to remove edge (w, t) and add edge (w', t) without creating multi-edges or self-loops. Since a NeighborSwitch removes exactly one edge (w, t) and adds exactly one edge (w', t), the number of edges between nodes of degree k (*i.e.* $w, w' \in V_k$) and nodes of degree l (*i.e.* $t \in V_l$) will remain the same, before and after the switch, therefore the value of $JDM(k, l)$ will not change. The NeighborSwitch is JDM preserving, and no updates to JDM are needed. □

Lemma 2.3 guarantees that, if construction has not terminated, there will always be at least one suitable t (neighbor of w but not neighbor of w') to perform the NeighborSwitch. In case there are multiple such neighbors t, picking any one of those candidates to perform the NeighborSwitch will work, since they all preserve the JDM. Although the choice of eligible neighbor, t, to perform the NeighborSwitch does not affect the correctness of the algorithm, it may affect the exact (not asymptotic) running time and the properties of the resulting realization. In our implementation, we purposely pick one random such neighbor, to avoid introducing bias in the structure of realizations.

Running Time. The running time of `2K_Simple` is $O(|E| \cdot d_{max})$, *i.e.* linear in the number of edges.

Proof. In each iteration of the `while` loop, one edge is always added, until we add all $|E|$ edges. However, we have to consider how much time it takes to pick nodes (v, w) and the cost of NeighborSwitch operations.

Naively chosen node pairs would become an issue for dense graphs, since there could be NeighborSwitches that remove previously added edges or add edges between the two degree groups. A simple solution is to keep track of $JDM^{\odot}(k,l) - JDM(k,l)$ many node pairs where edges can be added in a set P. For every pair of k, l, it is possible to initialize P by passing through at most $JDM^{\odot}(k,l)$ node pairs. A new (v,w) node pair can simply be chosen as a (random) element from P. If a neighbor switch for $v \in V_k$ (and similarly to w), rewires a neighbor $t \in V_l$, then $P = P \setminus (v',t) \cup (v,t)$ maintains available node pairs in P. Note: (v',t) might not be in P. This ensures that $|P| \geq JDM^{\odot}(k,l) - JDAM(k,l)$. These simple set operations can be done in constant time, and building P takes $O(E+V)$ time over all partition class pairs. Finally we remove (v,w) from P once the edge is added.

It takes $O(d_{max})$ time to choose a neighbor, t, of a node without free stubs, v, for NeighborSwitch, because the sets of neighbors can be at most $|d_{max}|$ and set difference takes linear time in the size of sets. Keeping track of nodes with free stubs allows us to pick v' for NeighborSwitch in constant time. In the worst case, there are at most two NeighborSwitches per new edge, hence the running time is $O(|E| \cdot d_{max})$. □

A tighter upper bound can be obtained by counting the running time of a NeighborSwitches for each degree group. We can express the number of edges E as a sum of stubs attached over nodes in each degree group k: $|E| = \sum_k D_k \cdot k/2$. In the worst case that each of these stubs will need a neighbor switch during an edge addition, the running time would be $O(\sum_k D_k \cdot k \cdot (k-1)/2) = O(\sum_k D_k \binom{k}{2})$, which is the number of paths of length two in the graph.

Space complexity. The input size proportional to the number of nonzero elements of the JDM, that is $O(d_{max}^2)$. The algorithm produces a graph that requires $O(V+E)$ space, while using temporary data structures. Therefore, `2K_Simple` uses the minimum space as it is required to store the final output graph. The details of the space requirements are as

follows.

`2K_Simple` requires constant look up time for nodes with free stubs, this can be achieved by storing an array of sets, where each set contains the nodes with free stubs for a given degree group. The size of this data structure is initially $O(V)$ and decreases over the execution of the algorithm.

As discussed in the proof, `2K_Simple` also maintains pairs of nodes (set P) for candidates to add edges. The size of P is $O(JDM(k,l))$ for a given k,l degree group pair and $O(\sum JDM) = O(E)$ over all iterations of the algorithm. (This is easiest to see in the example of targeting k-regular graphs, where P stores a set of $|E|$ candidate pairs initially, and we can keep track of nodes with free stubs in a single set of size $O(V)$.)

2.2.3 Connections to Related Work

`2K_Simple` adds an edge at a time while maintaining the following invariant for every k,l : $JDAM(k,l) < JDM^{\odot}(k,l)$ and $\forall v \in V_k : deg(v) \leq k$. This idea was also presented independently in [4]. Another approach, followed by [4], [63] and [16], is to add all edges between V_k, V_l according to $JDM^{\odot}(k,l)$ without considering degrees. This could create nodes with higher degree than their assigned degree group requires, which can then be resolved by using NeighborSwitch operations. Special realizations with the BDI property can also be constructed using these ideas: [4] adds further restriction to the first approach such that at every edge addition the BDI property is maintained, while [63] provides an algorithm using the second approach, where no NeighborSwitch operations are required.

The common idea behind all 2K construction algorithms is to add one edge at the time so as to not exceed node degrees or JDM; algorithms vary on if/how they violate any of these two properties and on when/how to correct it using local rewiring (NeighborSwitch in our

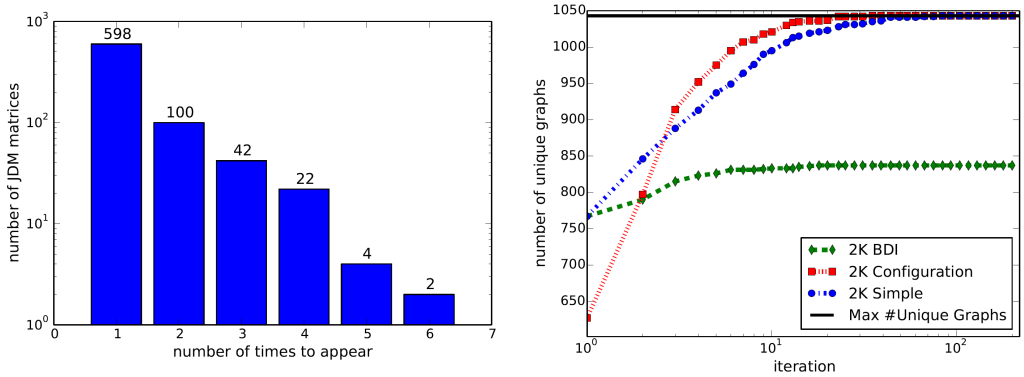
terminology); which is always possible for realizable JDMs. Please see Section 1.3.3 for a timeline. The particular order used in which `2K_Simple` adds and rewires edges is essential for being able to target additional properties, during construction, thus enabling a framework beyond just 2K.

`2K_Simple`'s run time complexity is comparable to other proposed 2K algorithms. Interestingly, it is even comparable to 1K construction algorithms that produce any 1K realization with non-zero probability [12, 18]. Indeed, $O|E|$ is the minimum required to construct a graph with $|E|$ edges.

2.2.4 Space of Realizations

The order in which `2K_Simple` adds edges is unspecified. The algorithm can produce any realization of a realizable JDM, with a non-zero probability. Considering all possible edge permutations as the order in which to add the edges, the ones where no neighbor switch is required correspond to all the possible realizations. Unfortunately, the remaining orderings are difficult to quantify, thus the current algorithm cannot sample uniformly from all realizations with a target JDM during construction. We experimentally show that `2K_Simple` can construct *all* possible graphs with up to seven nodes, while `2K_BDI` can produce much less.

We use the library NetworkX [20] to generate all 1044 non-isomorphic graph instances that contain seven nodes. We should note that the number of such graph instances increases exponentially with size e.g. for n=24 it is $\sim 1.95 \times 10^{59}$ [1]. For this reason we use in our experiment a small size of n=7 that gives 1044 instances. For each graph instance we calculate the corresponding JDM, which results in 768 unique JDM matrices (because there are cases where several graph instances correspond to the same JDM matrix). Fig. 2.3(a) shows the frequency of JDM matrices. We see that 598 matrices appear only once. Therefore,



(a) Frequency of JDM matrices.

(b) Cumulative number of unique graphs with seven nodes.

Figure 2.3: Generation of all non-isomorphic graph instances with 7 nodes.

if a generator received as input one of those JDM matrices it would always produce the same graph. On other extreme, two JDM matrices appear 6 times each. Therefore, if a generator received as input one of those JDM matrices it could produce either of the 6 corresponding graphs.

We conduct the following experiment. In each iteration, we feed the three algorithms with all 768 JDM matrices and we observe how many unique matrices each algorithm has generated, cumulatively since the first iteration. Fig. 2.3(b) shows the results. We observe that both simple graph construction algorithms (`2K_Simple` and `2K_BDI`) generate 768 unique graphs in the first iteration; `2K_Configuration` generates less graph instances due to multi-edges, which we removed. As the number of iterations increases, we observe that both `2K_Simple` and `2K_Configuration` reach 1044 (*i.e.* the total number of unique graphs corresponding to the 768 unique JDMs) in less than 100 iterations. However, the `2K_BDI` algorithm is unable to create more than 837/1044 (=80%) unique graphs after 200 iterations, due to the BDI constraint, discussed in Section 1.3.

Fortunately, once one realization is constructed (using `2K_Simple`), it is possible to sample from the space of all realizations, using edge-rewiring. In particular, it has been shown that JDM realizations are connected via 2K-preserving double-edge swaps [16],[4]. This method

is typically used by MCMC approaches that transform one realization to another by rewiring edges so as to preserve target properties.

In the next subsections, we exploit the flexibility of `2K_Simple` and we extend it to target additional properties, in addition to $JDM^{\odot}$. In Section 2.3.1 we control (approximately) the average clustering by controlling the order in which edges are added. In Section 2.3.3 we impose (exactly) node attributes by exploiting the flexibility in the number of degree groups with the same assigned degree. In Section 2.3.4 we consider the space of realizations with a target number of connected components.

2.3 2K with additional constraints

2.3.1 2K+S: Target JDM and Clustering

We proved that the realizability of JDM and a fixed number of triangles is NP-Complete [19] in Section 2.3.2. This motivated us to design efficient heuristics that target different notions of clustering (namely 2.25K when average clustering $\bar{c}^{\odot}$, is targeted and 2.5K when degree-dependent clustering $\bar{c}^{\odot}(k)$, is targeted).

MCMC Approach. In the original dK-series paper [52], 2K-preserving 3K-targeting was attempted via the classic JDM-preserving double edge swap as follows. Starting from a JDM realization, randomly select edges (a, b) and (c, d) such that $deg(a) = deg(c)$ and a,b,c,d are four distinct nodes (to avoid self-loops) and (a,d), (b,c) are not present before rewiring (to avoid multi-edges), as in Fig. 2.4. If $deg(a) = deg(c)$ then the swap obviously preserves the JDM of the graph. It is referred to as **JDM-preserving double-edge swap** and it is used in MCMC to transform graph G to other realizations G' with the same JDM, while targeting other properties. Here, we perform a double-edge swap iff it brings the graph closer to the

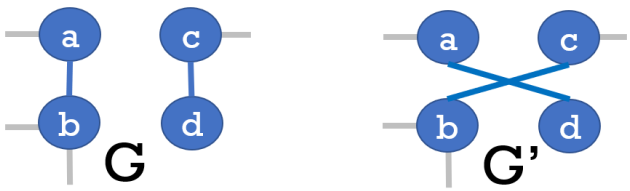


Figure 2.4: A JDM-preserving double-edge swap is a rewiring of edges (a,b), (c,d) to (a,d), (b,c), where a,b,c,d are four distinct nodes (to avoid self-loops) and (a,d), (b,c) are not present before rewiring (to avoid multi-edges). If $deg(a) = deg(c)$ then the swap obviously preserves the JDM of the graph. It is referred to as **JDM-preserving double-edge swap** and it is used in MCMC to transform graph G to other realizations G' with the same JDM, while targeting other properties.

target 3K (according to a well-defined distance metric), accept the rewiring. Unfortunately, this happens with a very small probability and the naive MCMC approach was very slow in practice, taking weeks or months to produce a single realization for large graphs.

We improved the 2K-preserving clustering-targeting MCMC by carefully selecting candidate edges to swap so as to control the number of triangles: select edges with low number of shared partners to create triangles, select random edges to destroy triangles. The rationale is that it is easier to destroy than create triangles. Although this reduced the running from weeks to days we still faced scalability problems.

Construction Approach. We modify `2K_Simple` so as to control the order in which edges are added and create the target clustering *during* the 2K construction, not with MCMC after that. Let E' be any permutation (order) of possible node pairs $\{v, w\}$. We follow the order in E' when we consider adding edges in Algorithm 2.1, line 3: if two node pairs $E'_i = (v_i, w_i)$ and $E'_j = (v_j, w_j)$ are s.t. $i < j$, then edge (v_i, w_i) will be considered for addition (line 5 in `2K+S`) before (v_j, w_j). The key question is: what is the right order E' of adding edges so as to control clustering?

Figure 2.5 depicts our approach. We assign every node $v \in V$ to a coordinate r_v randomly selected from a one-dimensional coordinate system $(0, 1)$. We define the distance of v and w as $dist(v, w) = \min(|r_v - r_w|, 1 - |r_v - r_w|)$. If we add edges in increasing distance,

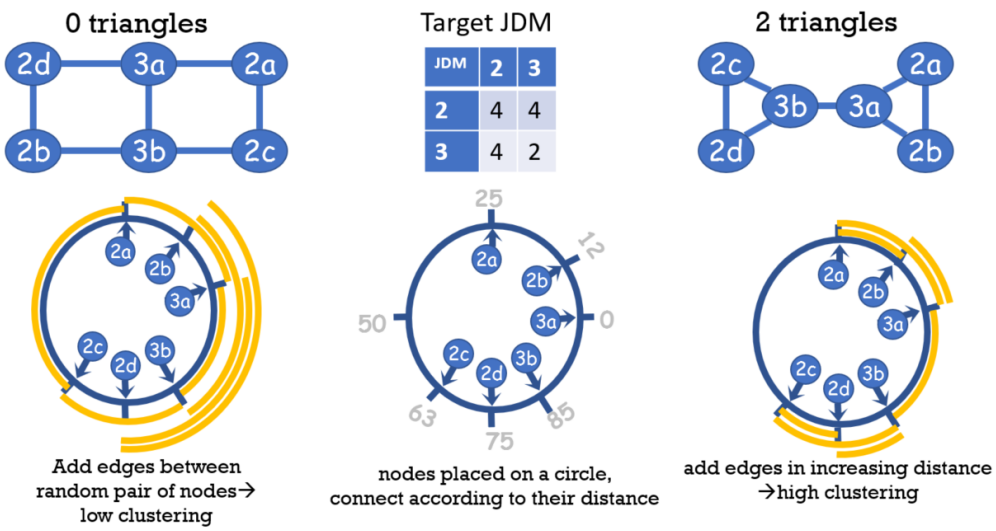


Figure 2.5: Approach for targeting clustering during 2K construction. `2K_Simple` runs with the target JDM, but we control the order in which to add edges, this results in either low (right) or high (left) clustering.

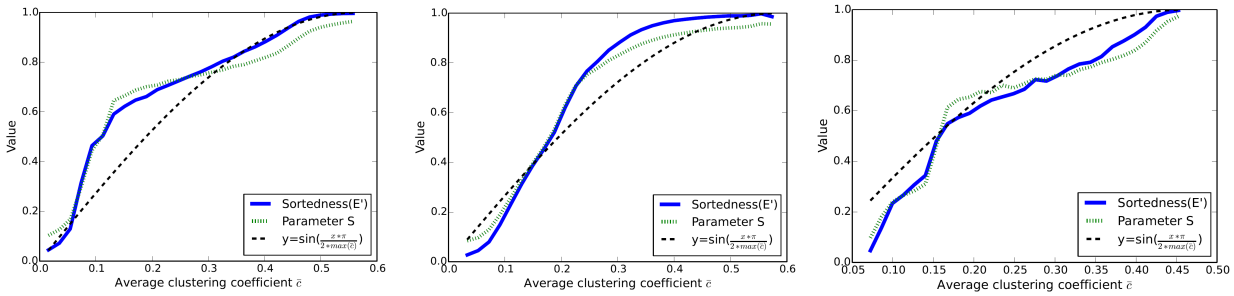
we connect nodes near each other, thus creating many triangles among nearby nodes in the coordinate system (as on the right side of the figure). If we add edges in random order, we create very few triangles (as shown on the left side of the figure). If we control the fraction of edges that are added in increasing distance vs. at random, we can control the clustering.

2.25K: Targeting JDM and Average Clustering

We introduce the notion of sortedness, S, to control the order of E' in which we add edge during graph construction. Two node pairs E'_i and E'_j are inverted in an order E' iff $(i < j)$ *and* $dist(v_i, w_i) > dist(v_j, w_j)$. We define the *sortedness* of a list E' as the fraction of non-inverted node pairs:

$$S = \text{sortedness(E')} = 1 - \frac{\text{number of inversions in list E'}}{|E'|(|E'|-1)/2} \in [0,1]. \tag{2.2}$$

We experimented with the effect that an order of node pairs E' has on the structure of the generated graph and we found that the sortedness S is positively correlated with the average



(a) Erdős-Rényi generator, $n = 100, m = 483, p = 0.1$

(b) Powerlaw-Cluster generator, $n = 100, m = 196, p = 0.1$

(c) Facebook Caltech network, $n = 769, m = 16,656$

Figure 2.6: Relation of average clustering coefficients, $\bar{c}$ and sortedness parameter, S.

clustering coefficient, $\bar{c}$, of the graph; see details on tuning parameter S. Fig. 2.6 shows an example of the correlation between the sortedness parameter, S, and average clustering coefficient for three types of graph inputs: two graph models (Erdős-Rényi, Powerlaw-Cluster[44]) to generate two graph instances and we select one real-world network, Facebook Caltech [69]. For each graph instance, with n nodes and m edges, we calculate its JDM and set it as our target. We then generate 10^5 graphs with sortedness and parameter S values varied between [0,1], and record the average clustering coefficient $\bar{c}$ of each generated graph. Fig. 2.6 shows that, for a fixed $JDM^{\odot}$, by setting parameter S between 0 and 1, we can also control the average clustering coefficient $\bar{c}^{\odot}$ of the produced graph between $min(\bar{c})$ and $max(\bar{c})$. We use function $y = sin(\frac{x \cdot \pi}{2 \cdot max(\bar{c})})$ to approximate the observed relation between parameter S and $\bar{c}$. Therefore, setting parameter $S = s$ roughly corresponds to an average graph instance with average clustering coefficient equal to $\frac{2 \cdot max(\bar{c}) \cdot arcsin(s)}{\pi}$. It is also intuitively expected [33], that values of sortedness(E') close to 0 produce graph instances with minimum clustering over all graph instances on average. Values of sortedness(E') close to 1 produce graph instances with maximum clustering over all graph instances on average.

Algorithm. Next, we present an algorithm to achieve exactly a target $JDM^{\odot}$ and approximate clustering by controlling the sortedness S of adding nodes. In the first stage, it attempts to add edges using a given order E' of all possible node pairs defined by $order(List, S)$. The

Algorithm 2.3 `2K+S`

Input: $JDM^{\odot}, S$
Stage 1:
1: $E = \{\}$
2: $E' = order(\{(v,w) : \forall v,w \in V\}, \text{sortedness} = S)$
3: **forall** $\{v,w\} \in E'$ **do**
4: $v \in V_k$, $w \in V_l$
5: **if** $\text{JDM}(k,l) < \text{JDM}^{\odot}(k,l)$ **and** $deg(v)k$ **and** $deg(w)l$ **do**
6: $E \leftarrow E \cup \{v,w\}$
7: $JDM(k,l)++$; $JDM(l,k)++$
Stage 2:
8: **if** $\sum \text{JDM}(k,l) \sum \text{JDM}^{\odot}(k,l)$ **do**
9: Finish graph construction using `2K_Simple`

function $order(List, S)$ (used in line 2 of Algorithm 2.3) determines the order of node pairs E', that will be considered for addition, so as to (approximately) set as S the sortedness of the input $List$.

Similarly to `2K_Simple`, it only adds edges if the current $JDM(k,l)$ value does not exceed the target $JDM^{\odot}(k,l)$. Differently than `2K_Simple`, it has an additional constraint: it adds edges between two nodes (v,w) *only if* there are free stubs to connect the nodes ($deg(v) < k$, $deg(w) < l$). Therefore at the end of the first stage, despite considering all possible node pairs, there might be some nodes with free stubs, since we do not allow multi-edges or self-loops. In Stage 2, we use algorithm `2K_Simple`, starting from the partially built graph at the end of Stage 1: we add edges between any remaining nodes with free stubs and complete the graph. It follows directly from the properties of `2K_Simple`, that this will produce the exact $JDM^{\odot}$.

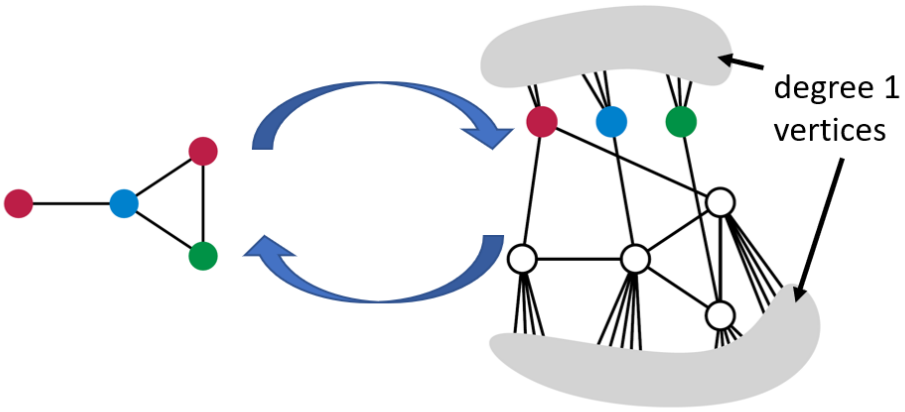
Running Time. The time complexity of `2K+S` is similar to `2K_Simple` for adding edges (*i.e.* $O(|E| \cdot d_{max})$), plus the time for the function order($List$, S). If naively implemented, the time to sort the input list E' is $O(|E'|log(|E'|))$. However, the list E' is consumed by lines 4-7 in Algorithm 2.3), which require at most $|E|$ edges that pass the condition in Line 5. Therefore, we argue that we do not need to enumerate all elements of E' because only some of the node

pairs in E' will not be rejected by the condition in Line 5. In practice, we observed that enumerating the first $k|E|$ node pairs of list E', where k is some small number, suffices to add the overwhelming majority of edges in the graph. The small number of remaining edges (if any) will be taken care of by Stage 2. Furthermore, we use the coordinate system r_v to sort nodes in each degree group k which takes time $O(\sum_k D_k log(D_k))$. After this initial sorting phase, each node v can find its closest k neighbors in linear time. In summary, the running time of a smart implementation of the function order($List$, S) that returns $k|E|$ elements is $O(k|E| + \sum_k D_k log(D_k))$. The expression is dominated by the term $k|E|$ in real-world graphs, which makes the running time approximately linear in the number of edges.

Space Complexity. Naive implementation of `2K+S` would require to generate all possible edges taking $O(|V|^2)$ space. An improved implementation only adds $O(|E|)$ edges which reduces the space complexity to the overall $O(|V| + |E|)$.

2.5K: Target $JDM^{\odot}$ and Degree-Dependent Clustering

Targeting $\bar{c}^{\odot}(k)$ is even more challenging than targeting $\bar{c}^{\odot}$. Our intuition from MCMC was that it is difficult to find a double edge swap that creates triangles while it is easier to destroy triangles. Therefore, we propose to (1) create a 2K realization with many triangles (such as a `2K+S` with $S = 1$) and (2) use the improved MCMC described above to destroy triangles. This indeed worked well in practice in terms of targeting $\bar{c}^{\odot}(k)$ and running time. This idea is evaluated in Section 2.5.



JDM	1	4	1*4+1+1	2*4+3+1	3*4+2+1	4*4+2+1
1	0	8	4	8	12	16
4	8	0	1	1	1	1
6	4	1	0	1	0	0
12	8	1	1	0	1	1
15	12	1	0	1	0	1
19	16	1	0	1	1	0

Figure 2.7: On the left is a graph with a coloring. On the right is a realization of JDM-Color$_G$ where the edges from the coloring nodes represent the previous coloring and the degree one nodes are represented by clouds. On the bottom we show the JDM-Color$_G$ for the example graph.

2.3.2 2K+#△: NP-Hardness for JDM with fixed number of triangles

It is possible to realize graphs with a target JDM as forests [66, 68] or trees [5] in polynomial time. In this section, we show that realizability of the JDMs of graphs with no triangles is much harder. We define this problem as the following:

Problem 2.1. 2K+#△-REAL

INPUT: *a JDM and a number $t \in N$ of triangles*

OUTPUT: *yes if there is a graph that realizes the JDM and has t triangles, no otherwise*

Given a graph G with n nodes we will create a JDM with $poly(n)$ nodes that can be realized with the same number of triangles as G if and only if G is 3-colorable. The key idea is that every realization of this JDM will contain a copy of G as a subgraph and we use additional nodes and the number of triangles to represent valid coloring.

For a graph G, we construct JDM-Color$_G$ as depicted in Figure 2.7 above. First the degrees we will use are $d_i = s_i + deg(v_i) + 1$ where $s_i = (i+1)n$ for each node $v_i \in G$. Because the $deg(v_i)$ are all less than n, we know $s_i < d_i < s_{i+1}$ and so each node in G corresponds with a unique degree in JDM-Color$_G$. We also use two additional degrees: 1 and n. We set the constraints as follows:

Edge constraints, for each edge (v_i, v_j) in G:

$$JDM(d_i, d_j) = 1 \tag{2.3}$$

Coloring constraints, to create coloring gadget we use degree n:

$$\forall v_i \, JDM(n, d_i) = 1 \tag{2.4}$$

Equal degree constraint, to create equal degree for coloring nodes:

$$JDM(n, 1) = 2n \tag{2.5}$$

Unique degree constraints, to create unique degrees for each node v_i:

$$\forall v_i \, JDM(d_i, 1) = s_i \tag{2.6}$$

The remaining entries of JDM are all set to zero. An example is shown for the construction of JDM-Color$_G$ in Figure 2.7.

Lemma 2.4. *In any realization of* JDM-Color$_G$*, there is exactly one node for each degree d_i, there are $2n + \sum s_i$ nodes with degree one, and 3 nodes with degree n.*

Proof. We can compute the number of nodes with each degree: $\forall i, |V_i| = \sum_{j=1} \frac{JDM(i,j)}{i}$.

The nodes with degree d_i have an edge for each neighbor of v_i in G totalling $deg(v_i)$ such edges, 1 edge to the coloring gadget from $JDM(n, d_i) = 1$, and several edges to degree one nodes from $JDM(d_i, 1) = s_i$. All together these sum to exactly d_i. Therefore there is only one node with degree d_i.

nodes with degree one appear only in $JDM(n, 1)$ and $JDM(d_i, 1)$ for each d_i, summing shows that there are $2n + \sum s_i$ such nodes.

The nodes with degree n have n edges to the nodes in G and $2n$ edges to degree one nodes from $JDM(n, 1)$. Since their degree is n, the number of such nodes is $3n/n = 3$. □

We refer to the three nodes of degree n as the coloring nodes.

Lemma 2.5. *If G is 3-colorable, then* JDM-Color$_G$ *is realizable.*

Proof. Create a new graph G' as follows:

- For each node v_i in G, create a node v'_i and connect v'_i to s_i new degree one nodes
- For each edge u, v in G, connect u' and v' with an edge
- Add $2n$ isolated nodes
- Add three new nodes r, g, and b

- For each red node v in the 3-coloring of G, connect v' to r
- For a green node v in the 3-coloring of G, connect v' to g
- For a blue node v in the 3-coloring of G, connect v' to b
- Connect each isolated node to one of r, g, and b such that $deg(r) = deg(g) = deg(b) = n$

Since each node in G' only connects to one coloring node, their original neighbors, and s_i degree one nodes, nodes v_i will have $d_i = s_i + deg(v_i) + 1$. The nodes r, g, and b each have degree n. The remaining nodes of G' all have degree one.

To show that G' satisfies the edge constraints of the JDM, we now correspond the edges of G' with the entries in JDM-Color$_G$. The edge between the nodes of degrees d_i and d_j occurs between v'_i and v'_j. Every node v' in G' has one edge to a coloring node and the coloring nodes have a total of $2n$ degree one neighbors. Further for each v'_i, the node of degree d_i, has s_i neighbors of degree one. This accounts for all the edges in G' and each group of edges is now identified with an entry of the JDM.

Since G had a valid coloring, the edges from nodes in v' to coloring nodes can not participate in any triangles. Also degree one nodes are not able to form triangles. So the only triangles in G' are copies of triangles in G. Thus G' is a realization of JDM-Color$_G$. □

Lemma 2.6. *If* JDM-Color$_G$ *is realizable, then* G *is 3-colorable.*

Proof. Suppose there is a graph G' that realizes JDM-Color$_G$. Because of Lemma 2.4, G' has n nodes with the degrees d_i, 3 nodes with degree n, and $2n + \sum s_i$ nodes with degree 1. For every edge v_i, v_j in G, there is an edge in G' between the unique nodes of degree d_i and d_j, thus G' has a copy of G as a subgraph.

Since the number of triangles in G and G' are equal and G' has a copy of G, there are no triangles involving the coloring nodes.

Associate the colors *red*, *green*, and *blue* with the three degree n nodes in G'. We can assign each v_i the color of the degree n node neighbor of the degree d_i node. Two adjacent nodes in G cannot be assigned the same color otherwise there is a triangle between the two corresponding nodes in G' and the coloring node. So G has a proper 3-coloring. □

Based on these two lemmas together, we can conclude:

Theorem 2.1. 2K+#△-REAL *is* NP*-Complete.*

Proof. By Lemma 2.5 and Lemma 2.6, 2K+#△-REAL is NP-Hard. It is also in NP because a graph that realizes a distribution and has the given number of triangles is a verifiable witness by simply counting triangles and computing the JDM. □

There are triangle free graphs where 3-Coloring remains NP-Hard [51], thus our reduction also includes the case to realize JDMs as triangle free graphs.

JDM is a special case of Joint-Degree and Attribute Matrix (JDAM) [34] and Partition Adjacency Matrix (PAM) [24]. The JDAM problem relaxes the constraint for every V_k to have nodes with different degree. PAM problem relaxes this problem such that every part of the partition has an arbitrary degree sequence, instead of nodes with the same degrees. Realizability of JDAM and PAM problems with fixed number of triangles remains NP-Complete.

NP-Hardness of 2K+Clustering

In this section, we consider implications of Theorem 2.1 targeting to clustering of 2K realizations. We consider the clustering coefficients defined in Section 1.3.3: global clustering, local clustering, average clustering and degree-dependent clustering.

The number of two-paths (simple paths of length two) in every realization of a given JDM are equal, since realizations have the same degree distribution that specifies this number. The realizability of JDM with fixed c_G and JDM with fixed number of triangles are equivalent problems and our previous result directly applies to this problem.

The realizability of JDM with fixed local, or degree-dependent clustering coefficients are equivalent to the realizability of JDM with the number of triangles assigned to each node or degree group respectively. We can easily apply the previous proof strategy to these problems with a proper assignment of triangles. The key observation is that during our reduction we create degree groups with a single node ($|V_{d_i}| = 1$) for every node, v_i, in G. The assignment of number of triangles is equivalent to the number of triangles in G for each node. Degree 1 nodes can not form triangles and degree n (coloring) nodes are not allowed to form triangles to ensure valid coloring during the reduction. For both cases we can assign 0 triangles either per degree group or for each individual node. Our reduction still carries through with these modifications, which leads to the conclusion that realiability of JDM with fixed local or degree dependent clustering is NP-Complete.

Average clustering coefficient requires the consideration of the NP-completeness of 3-Coloring on triangle free graphs. As we have shown earlier, realizability of JDM with 0 triangles remains NP-Complete, hence realizability of JDM with $\bar{c} = 0$ is also NP-Complete.

2.3.3 2K+A: Targeting JDM and Node Attributes

JDM vs. JAM. JDM only describes correlations between the degrees of connected nodes. However, in many contexts, capturing correlations of node attributes in the network model better characterizes the graph [45, 21, 60]. For example, in social networks, the similarity of attributes between two nodes often affects the creation of an edge between them. We assume that there is a set of categorical attributes with p possible values. Each node $v \in V$ can be

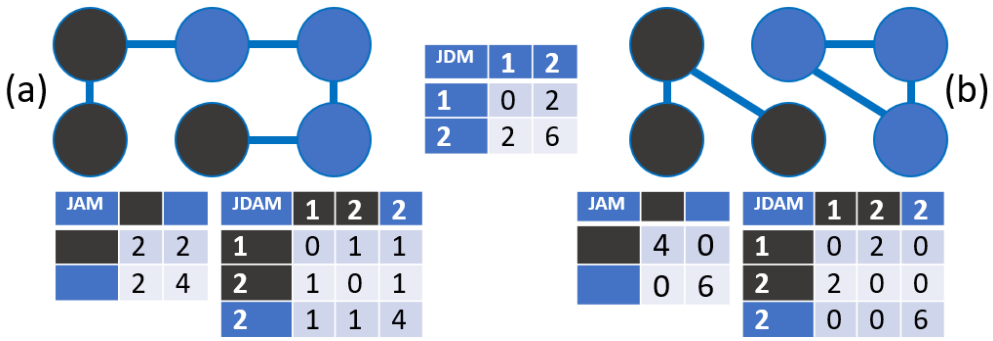


Figure 2.8: Example of two graphs, (a) and (b), with the same Joint Degree Matrix (JDM) and different Joint occurrence of Attributes Matrix (JAM) based on colors (black, blue) as attributes, thus different JDAMs.

assigned to only one categorical attribute. Let A_i be the set of nodes that have attribute i, for $i = 1, ..., p$. We can define the *Joint occurrence of Attributes Matrix (JAM)* as the number of edges connecting nodes in A_i with nodes in A_j.

$$\text{JAM}(i,j) = \sum_{v \in A_i} \sum_{w \in A_j} 1_{\{\{v,w\} \in E\}}. \tag{2.7}$$

However, JAM alone does not capture the network structure. In Fig. 2.8, we show a toy example. The graphs in Fig. 2.8(a) and Fig. 2.8(b) have the exact same JDM but different JAM. And conversely, examples of networks with the same JAM and different JDM can be constructed as well.

JDAM. We propose to incorporate correlations of node attributes on top of the JDM matrix as follows. If V_k is the set of nodes that have degree k for $k = 1, ..., d_{max}$ and A_i the set of nodes that have attribute i, for $i = 1, ..., p$, then let the degree-attribute group $B_{\{k,i\}} = \{v | v \in V_k, v \in A_i\}$ be the set of nodes that have degree k and attribute i. The number of degree-attribute groups is at most $d_{max} \cdot p$. We define the *Joint Degree and occurrence of Attributes Matrix* (JDAM) as the number of edges connecting nodes in $B_{\{k,i\}}$

Algorithm 2.4 `2K_Simple_Attributes`

Input: $JDAM^{\odot}$
Init: $JDAM(\{k,i\},\{l,j\}) = 0,\ \forall\ (\{k,i\},\{l,j\}) \in JDAM^{\odot}$
1: **for** $(\{k,i\},\{l,j\}) \in JDAM^{\odot}(\{k,i\},\{l,j\})$
2: **while** $JDAM(\{k,i\},\{l,j\}) < JDAM^{\odot}(\{k,i\},\{l,j\})$
3: Pick any nodes $v \in V_{\{k,i\}}$ and $w \in V_{\{l,j\}}$
s.t. (v,w) is not an existing edge
4: **if** v does not have free stubs:
5: v': node in $V_{\{k,i\}}$ with free stubs
6: NeighborSwitch(v,v')
7: **if** w does not have free stubs:
8: w': node in $V_{\{l,j\}}$ with free stubs
9: NeighborSwitch(w,w')
10: add edge between (v, w)
11: $JDAM(\{k,i\},\{l,j\})$++ ; $JDAM(\{l,j\},\{k,i\})$++
Output: simple graph with $JDAM = JDAM^{\odot}$

with nodes in $B_{\{l,j\}}$ for degree-attribute groups $\{k,i\}$ and $\{l,j\}$.

$$\text{JDAM}(\{k,i\},\{l,j\}) = \sum_{v \in B_{\{k,i\}}} \sum_{w \in B_{\{l,j\}}} 1_{\{\{v,w\} \in E\}}. \tag{2.8}$$

Example JDAMs are shown in Fig. 2.8. A JDAM is similar to JDM, but each row now describes not only a degree k but a degree-attribute pair $\{k,i\}$; and similarly for the columns.

Section 2.3.3 describes the algorithm for targeting a given JDAM and the details are provided below.

It turns out that `2K_Simple` can be gracefully extended to construct a simple graph with a target $JDAM^{\odot}$ as shown in Algorithm 2.4. We can observe that our proofs (and others from related work) depend on the fact that within a degree group degrees are equal among nodes, however these proofs do not have restrictions on how many times a degree group appears with the same degree. We can apply and trivially extend earlier results including sufficient and necessary conditions for realizability, construction algorithms, existence of BDI realizations, importance sampling algorithm extensions from JDM, connectivity of space of

realizations over $JDAM$ preserving double-edge swaps and MCMC properties.

The running time and space complexity analysis follows `2K_Simple`. The only change is that the JDAM input has a size of $O((d_{max} \cdot p)^2)$, where a sparse representation is better characterized by the number of non-zero JDAM entries or $O((\text{number of observed node degree and attribute combinations})^2)$.

2.3.4 2K+CC: Number of Connected Components

In this section, we target the number of connected components (CC), in addition to the JDM – a property which has not been explicitly targeted or characterized in the past. An algorithm for constructing graphs with a realizable JDM and a single CC, if such exist, has been provided in [4]. However, there may be JDM realizations with a different number of connected components k, s.t. $1 \leq k_{min} \leq k \leq k_{max}$. Our main result on this problem is the following:

Theorem 2.2. *The space of simple, undirected graphs with a target JDM and no more than $k^{\odot}$ number of CCs is connected under a sequence of JDM-preserving double-edge swaps.*

There are counterexamples that show that the above statement is not true for *fixed* k, *i.e,* realizations with *exactly* k CCs are not necessarily connected under double-edge swaps. However, we show that the JDM realizations with a number of CCs *up to a maximum target number,* $k \leq k^{\odot}$, is connected over double-edge swaps.

Figure 2.9 depicts how every pair of realizations, (A, B), can be reached via a sequence of JDM-preserving double-edge swaps where every intermediate realization has less than the maximum of A and B's number of CCs. Lemma 2.7 uses double-edge swaps that merge CCs and decrease k. Therefore both A and B can be transformed to A', B' with the same JDM and the minimum number of CCs, k_{min}. Lemma 2.8 guarantees that A' and B' can also be

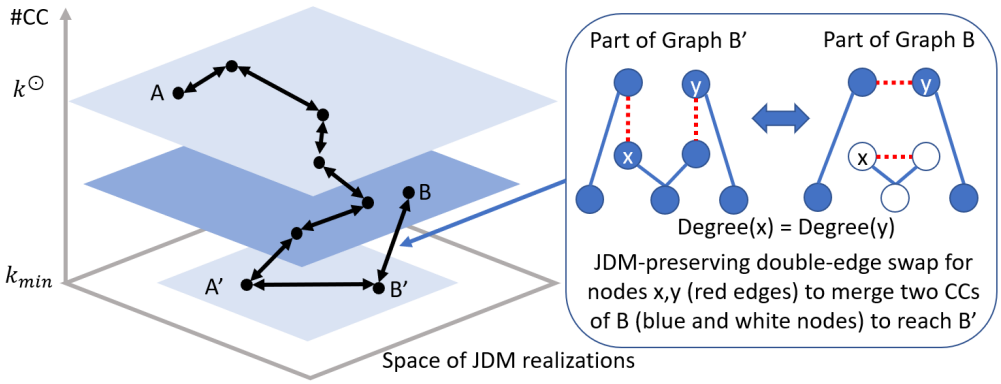


Figure 2.9: Space of JDM realizations with up to $k^{\odot}$ CCs is connected under JDM-preserving double-edge swaps (shown on right). A sequence of double-edge swaps (represented by arrows on left) exists to transform a graph realization A to B (or B to A), while using at most $k^{\odot}$ CCs.

transformed to each other via the same types of swaps.

Lemma 2.7. *There exists a JDM-preserving double-edge swap sequence that transforms any JDM realization to a realization with minimum number of CCs, such that there is no double-edge swap that increases the number of CCs.*

Lemma 2.8. *The space of JDM realizations with minimum number of CCs, k_{min}, is connected under JDM-preserving double-edge swaps.*

In the following subsections, we prove Lemma 2.7 and Lemma 2.8 and finally we show that there always exists a balanced and minimum number of connected component realizations (BDI) of any JDM.

Proof of Lemma 2.7

First, we show how to construct a realization with minimum number of connected components from any JDM realization using JDM-preserving double-edge swaps (while not increasing the number of connected components at any step). The main algorithm and the notation follows the Valid Tree Construction algorithm described by Amanatidis *et al.* in [4],

but the key difference is that the modified algorithm starts from a JDM realization (instead of a $V-1$ "valid" edges) and only uses JDM-preserving double-edge swaps to construct a realization with minimum number of connected components. This change will require some adjustments in the algorithm and in the certificate that is produced to show that there is no realization with less than c number of connected components.

We use the following notation:

- V_i is a degree group (nodes with degree i) $\mathsf{V}= \cup V_i$.
- $F \subset \mathsf{V}$, and A is a partition of F.

Define a weighted graph $G^{cert.}(V', E', w)$ with a node for each element A_i in A and one node for each $V_x \notin F$, assign the following edges and weights:

- If $\exists V_x \in A_i$ and $\exists V_y \in A_j$ such that $JDM(x, y) > 0$, then add edge (i, j) with weight $w(i, j) = 1$.
- If $V_x, V_y \notin F$, then add edge (x, y) with weight $w(x, y) = JDM(x, y)$.
- If $V_x \notin F$ and $JDM(x, x) > 0$ add self-loop with weight $w(x, x) = JDM(x, x)$.
- For any A_i, V_x, add an edge with weight $w(i, x) = \sum_{z:V_z \in A} JDM(z, x)$, if $w(i, x) > 0$.

Given a realization $G(V, E)$ of a JDM with minimum number of connected components ($c \geq 1$), we can construct $G^{cert.}(V', E', w)$ for every F, A combination, by collapsing nodes corresponding to A and V_i, then the following inequality holds $\sum_{e \in E'} w(e) \geq |A| + \sum_{V_i \notin F} |V_i| - c$ in the constructed weighted graph, that has at most c components. This follows from the existence of the realization. If there is partition F and c, where the above inequality doesn't hold, it is a certificate to show that there is no realization with c connected components.

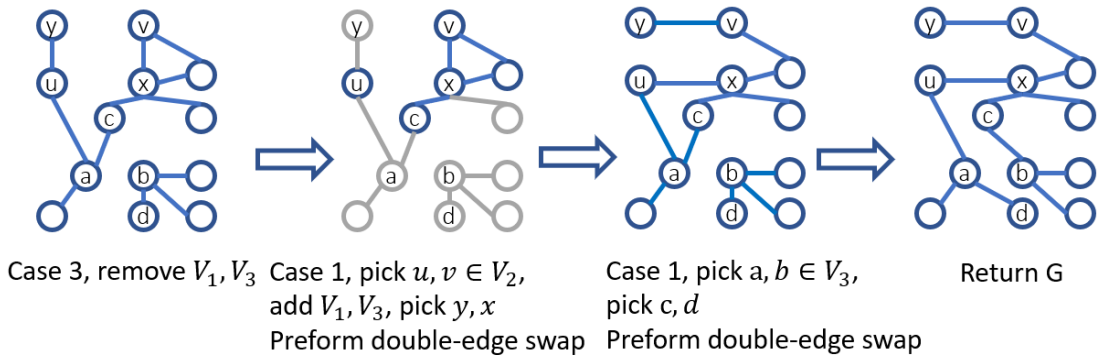


Figure 2.10: Example execution of Find-MinCC algorithm.

We refer to the modified algorithm from [4] as Find-MinCC here. Find-MinCC finds a swap sequence instead of building a partial realization of a spanning tree for input JDM. Fig. 2.10 shows a simple example to highlight the intuition behind how the algorithm uses double-edge swaps to move and break cycles to connect different connected components in a JDM realization.

Theorem 2.3. *The Find-MinCC algorithm finds a realization with the minimum number of connected components of JDM in polynomial time.*

Proof. It is trivial to show, that if G_0 is a forest or a single connected component realization, then the algorithm found a minimum number of connected component realization. Consider the three cases of Algorithm Find-MinCC.

First, we show that the algorithm terminates after polynomial many iterations: The recursion can only go to depth k - the number of distinct degrees - using **Case 3**, because at that point Case 2 will happen and the algorithm terminates. On the other hand, we will shortly show, that when Case 1 happens at depth j, then Case 1 will happen j consecutive times and the number of connected components will be decreased by one in G_0. This means that the while-loop can iterate at most $O(k|V|)$, using $2k$ iterations to merge pairs of connected components (loosely upper bounded by $|V|$).

Algorithm 2.5 Find-MinCC(G)

begin
V= $\cup_{i=1}^{k} V_i$; j=0; G=G_0
while G_0 is not connected or not a forest:
  begin
  $O_j = \{v : \text{nodes on cycles in } G_j\}$
  $C_j = \{V_i : O_j \cap V_i \neq \emptyset\}$
  $P_j = \{V_i : V_i$ intersects at least two connected components of $G_j\}$
  $Z_j = \{e \in G_j :$ at least one endpoint of e is in some $V_i \in P_j\}$
  Case 1: If $C_j \cap P_j \neq \emptyset$
    pick u, v in some $V_i \in C_j \cap P_j$ from different components in G and $u \in O_j \cap V_j$;
    j = max(j-1,0); $G_j = G \cup P_j \cup Z_j$; $G = G_j$;
    pick x: neighbor of u from a cycle in G;
    pick y: neighbor of v in G;
    remove xu, yv from G; add xv, yu to G;
  Case 2: else if $P_j = \emptyset$
    let $K_1, K_2, ..., K_\lambda$ be components of G_j
    let A_i =$V_x : V_x \subset V(K_i)$ for $1 \leq i \leq \lambda$;
    let $F = \cup_{i=1}^{\lambda} A_i$; let $A = \{A_i, ..., A_\lambda\}$;
    output (F, A); $G \cup_{i=0}^{j} P_i \cup_{i=0}^{j} Z_i$; terminate
  Case 3: else $G_{j+1} = G_j \setminus P_j; G = G_{j+1}; j = j + 1$
end
output G_0;
end

Similar to [4], we notice that if G_0 is not a realization with minimum number of connected components, then for any j such that a G_j is constructed by the algorithm we have $O_j \neq \emptyset$ (and thus $C_j \neq \emptyset$). G_0 either is a forest or contains cycles. Also any G_j if created will have $O_{j-1} \in V(G_j)$, because otherwise Case 1 would happen. Intuitively cycles are preserved as j increases.

Case 1. First, notice that Case 1 is exactly a JDM-preserving double-edge swap, thus any realization after performing these swaps will be a realization of the same JDM as the input graph. The argument is similar to [4]: the number of connected components will decrease by 1 in G_0, if Case 1 happens at G_j. Notice that two components that got connected of

$G = G_j$, K, K', must be subgraphs of the same connected components in G_{j-1} otherwise they would have been connected in an earlier iteration (*i.e.* when G was still G_{j-1}). After the double-edge swap, we call G'_{j-1} the graph that got back P_{j-1}, Z_{j-1} (*i.e.* the nodes and edges previously removed in at $j-1$ depth). The key observation is that, in G'_{j-1}, a new cycle is created for some v from $V_i \in P_{j-1}$, because v was on a path connecting K, K' in G_{j-1} before. This will result in another Case 1 in G'_{j-1} until j reaches 0 and results in a decrease of number of connected components by one in G_0. Notice that differently from [4], we first add back the P_{j-1}, Z_{j-1} and then perform the double-edge swap, in order to ensure that v has neighbors to perform double-edge swaps with at G'_{j-1}.

Case 2. The analysis of Case 2 is the same as Case 3 in [4], except we already know that all the required edges are present. Let K_1, K_2, K_λ be the connected components of G_j and let A and F be defined as in the algorithm. This assignment makes sense for G_j: if a node $v \in V_x$ is in K_i then all nodes of V_x are also in K_i (since $P_j = \emptyset$). If we consider the current graph, G_0, all cycles in G_0 are contained in the subgraph of G_0 induced by nodes of $\cup_i K_i$.

Following the proof from [4], we only have to notice that, if we identify all A_i with one single node to get H from G_0, H will have c connected components but contains no cycles. That means $|E(H)| = |V(H)| - c$, to have $c' = c - 1$ connected components it changes the above equality to $|E(H)| < |V(H)| - c'$, but we also know that $|V(H)| = |A| + \sum_{V_i \notin F} |V_i|$, and $|E(H)| = \frac{1}{2} \sum_{i:V_i \notin F} \sum_{j:V_j \notin F} JDM(i,j) + \sum_{i:V_i \notin F} \sum_{x=1}^{\lambda} \sum_{y:V_y \in A_x} JDM(i,y)$.

In conclusion, if we use (F, A) to construct the weighted graph $G^{cert.}(V', E', w)$ as before, then $\sum_{e \in E'} w(e) < |A| + \sum_{V_i \notin F} |V_i| - c'$, showing that no realization exists with less than c connected components. □ □

The running time of Find-MinCC will depend on the number of connected components of G and the minimum number of connected components achievable c. An input realization can be constructed in $O(k|E|)$ time. Each iteration in Find-MinCC can be easily done in

$O(|E| + |V|)$ time using Tarjan's bridge finding algorithm to identify nodes in cycles and using sets appropriately to do operations for O, C, P, Z. Find-MinCC runs in $O(k|V||E|)$ which dominates the input construction time. However, `2K_Simple` returns with realizations using c' connected components, this results in a better overall running time of $O(k(c'-c)|E|)$ and $O(k|E|)$ running time if we assume that $c' - c$ is some small constant. The Find-MinCC algorithm can be efficiently applied for realizations that have close to minimum number of connected components for a target JDM.

Since Find-MinCC only used JDM-preserving double-edge swaps without increasing the number of connected components at any step, the union of the swaps used during the algorithm will transform the input realization to a minimum number of connected component realization.

Proof of Lemma 2.8

In this section we show that every pair of JDM realizations with minimum number of connected components are connected over double-edge swaps. Again, this can be shown by using the proof for the space of JDM realizations from [4]. Here we have to make an additional constraint, such that during every double-edge swap the number of connected components will not increase.

The proof is based on induction on the size of the symmetric difference, k, between two JDM realizations with MinCC (G, G'). The key idea follows the results from [4], however, here we have to consider the number of connected components at every step. First we start by several simple observations and definitions about the problem.

We use a red-blue graph to describe the symmetric difference of edges where red represents edges only in G, blue represents edges only in G', black edges are present in both G, G' and edges not present in either graphs will be left empty. Red (blue) path is defined as a simple

path in the red-blue graph that only contains red (blue) and black edges.

There are two simple but crucial points, shown by Amanatidis *et al.* [4] and others in earlier work: (1) the number of red and blue edges for a node is the same and (2) there always exists a red-blue circuit decomposition for the red-blue graph using red and blue edges, *i.e.* the symmetric difference. Our proof is similar to proofs based on alternating red-blue cycles for degree sequences, but our cases also correspond to cases found in [4].

Similar to Amanatidis *et al.* [4], our proof is based on pairing nodes, that can be defined as two nodes from the same degree group on an alternating red-blue cycle of distance 2 (along the cycle). It is trivial to see that double-edge swaps around these nodes will be JDM-preserving.

There is only one case where the number of connected components would increase after performing a double-edge swap: $v - w - p - u - z$ (where p is the only (simple) path between w, u). A double-edge swap using w, z would return new paths $v - z$ and a new $w - p - u - w$ cycle, thus increasing the number of connected components. For simplicity, we will call these configurations red and blue "cut-paths" depending on whether they appear in red or blue graphs. However, degree 1 nodes cannot participate in a cycle, thus any double-edge swap using two degree one nodes will maintain the number of connected components. This observation will significantly shorten our proof since it means, that we can apply any available double-edge swap from Amanatidis *et al.* [4] to handle degree 1 nodes; for the rest of this discussion we assume that the pairing nodes have at least degree 2.

Double-edge swaps across different components cannot increase the number of connected components, only decrease if at least one edge was initially in a cycle. This observation also simplifies our proof, since double-edge swaps of this nature can be done without additional consideration of the connected components and solved by Amanatidis *et al.* [4].

The proof will show that starting from any symmetric difference ($k \geq 4$), we can reduce this

difference by at least 2 using double-edge swaps while both red and blue graphs preserve the number of connected components. In the main part of the proof, pairing nodes have degree at least two, thus when a cut-path exists, there is going to be a usable node either on the cut-path or another neighbor which we can pick. The following cases show that in every configuration (independently whether the neighbor has red, blue or black edge), we can progress to decrease k using double-edge swaps that preserve the number of connected components.

Base case $k = 4$: The red-blue graph will only contain a single alternating 4-cycle, $0 < k < 4$ is not possible. Since both realizations have the same CC, both red or blue double-edge swaps along the 4-cycle will maintain the number of connected components for both the red and blue graphs.

We break our cases into 3 main groups depending on the availability of pairing nodes and the length of alternating cycles where pairing nodes exist:

Case 1: If $k > 4$ and there are pairing nodes (u,v) in a 4-cycle: we have to consider red and blue cut-paths of the type discussed earlier around these nodes. There are two configurations of red and blue cut-paths that would cause an increase in CC if naive double-edge swaps were executed. This means that u,v do not share neighbors in the same colored graph.

Subcase 1: Red and blue cut-paths meet at a non-pairing node (x): depending on whether the first edge on blue cut-path from u to u_1 is blue or black we have to cases that we can handle similarly as shown in Fig. 2.11.

If (u, u_1) is blue, perform blue double-edge swap $(u, u_1), (v, x) \rightarrow (v, u_1), (u, x)$, this results in (v, u_1) is blue or black depending whether (v, u_1) was red or empty before; and (u, x) becoming black from red. In addition, we can perform another red double-edge swap $(u, u_2), (v, w) \rightarrow (v, u_2), (u, w)$ that behaves the same way as the first swap.

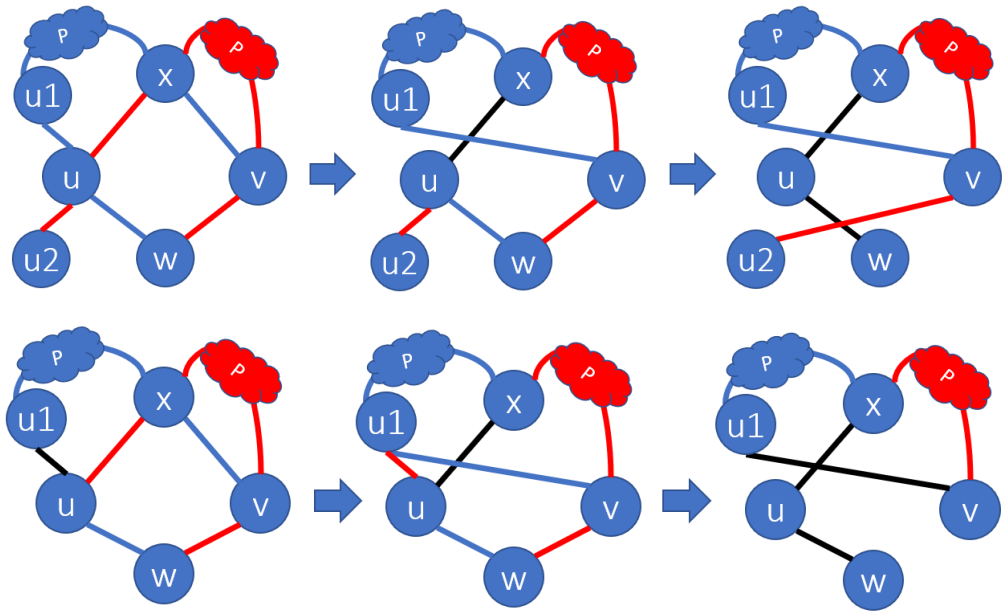


Figure 2.11: Case 1, subcase 1: (u, u_1) is blue (top), (u, u_1) is black (bottom).

If (u, u_1) is black, perform blue double-edge swap $(u, u_1), (v, x) \to (v, u_1), (u, x)$, this results in (v, u_1) is blue (in this case (v, u_1) has to be empty before); (u, u_1) becoming red from black; and (u, x) becoming black from red. In addition, we can perform another red double-edge swap $(u, u_1), (v, w) \to (v, u_1), (u, w)$ that removes all the red and blue edges.

Subcase 2: Red and blue cut-paths meet at a pairing node (v): since no paths cross the other pairing node u, and u has degree at least 2, there will be either two neighbors (one with red u_2 and one with blue edge u_1) or a neighbor with black edge. This case is very similar to the previous subcase 1, and swaps are shown in Fig. 2.12.

If (u, u_1) is blue, perform blue double-edge swap $(u, u_1), (v, x) \to (v, u_1), (u, x)$, this results in (v, u_1) is blue or black depending whether (v, u_1) was red or empty before; and (u, x) becoming black from red. In addition, we can perform another red double-edge swap $(u, u_2), (v, w) \to (v, u_2), (u, w)$ that behaves exactly the same way as the first swap.

If (u, u_1) is black, perform blue double-edge swap $(u, u_1), (v, x) \to (v, u_1), (u, x)$, this results in (v, u_1) is blue (in this case (v, u_1) has to be empty before); (u, u_1) becoming red from black;

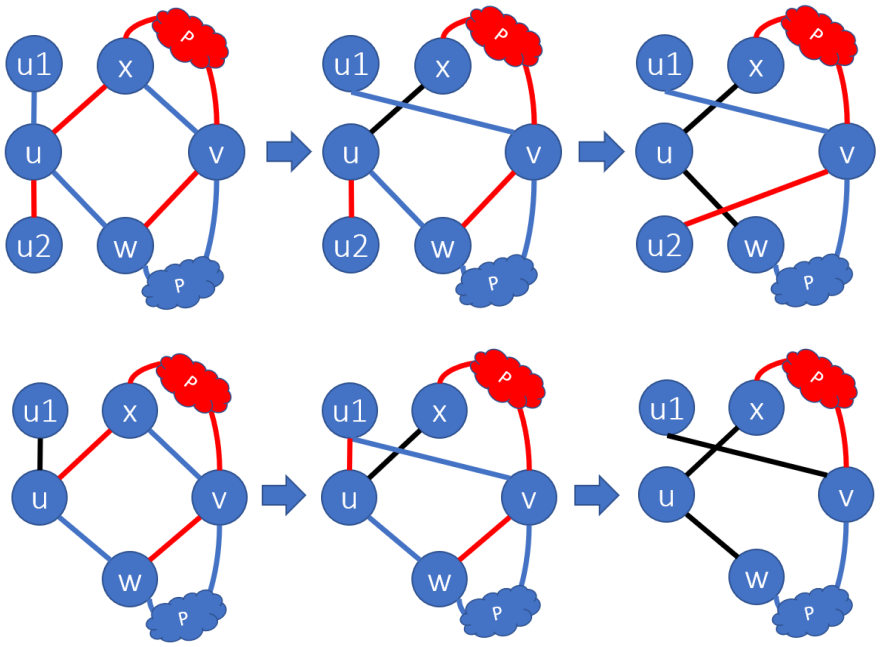


Figure 2.12: Case 1, subcase 2: (u, u_1) is blue (top), (u, u_1) is black (bottom).

and (u, x) becoming black from red. In addition, we can perform another red double-edge swap $(u, u_1), (v, w) \to (v, u_1), (u, w)$ that removes all the red and blue edges.

In all of these subcases the double-edge swaps will decrease k by 4, because we resolve the 4-cycle without creating more red or blue edges. More importantly, these double-edge swaps will not increase the number of connected components in neither the red nor the blue graph.

Case 2: If $k > 4$ and there exists pairing nodes (u,v) on an alternating cycle with length more than 4: We are only interested in the two neighbors along the cycle for each node u,v: x,y and the shared one: w. Here we have 3 major subcases depending on whether 0,1 or 2 black edges are formed along the cycle:

Subcase 1: there are no black edges: u, v have at least one neighbor difference, depending on whether it is y or another neighbor v_1 there are different cases, as follows:

If v has a blue neighbor, v_1, not on the cycle: we can perform a blue double-edge swap $(v, v_1), (u, w) \to (v, w), (u, v_1)$; this makes (v, w) black and (u, v_1) blue ((u, v_1) was not red

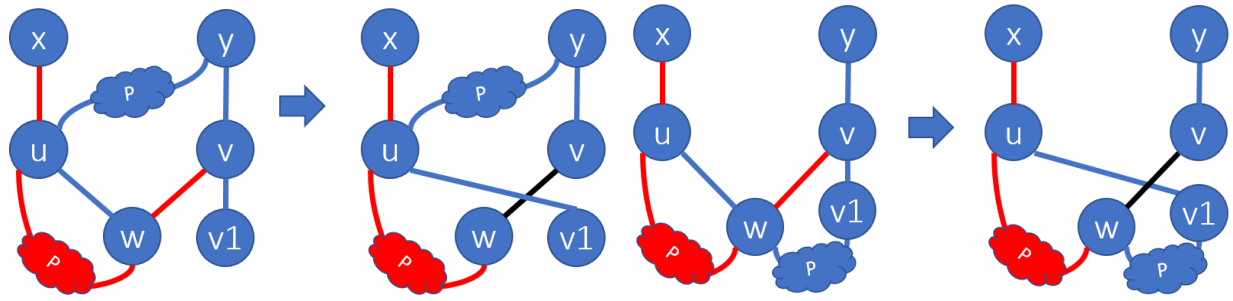


Figure 2.13: Case 2, subcase 1: (v, v_1) is blue and two configurations of blue cut-paths.

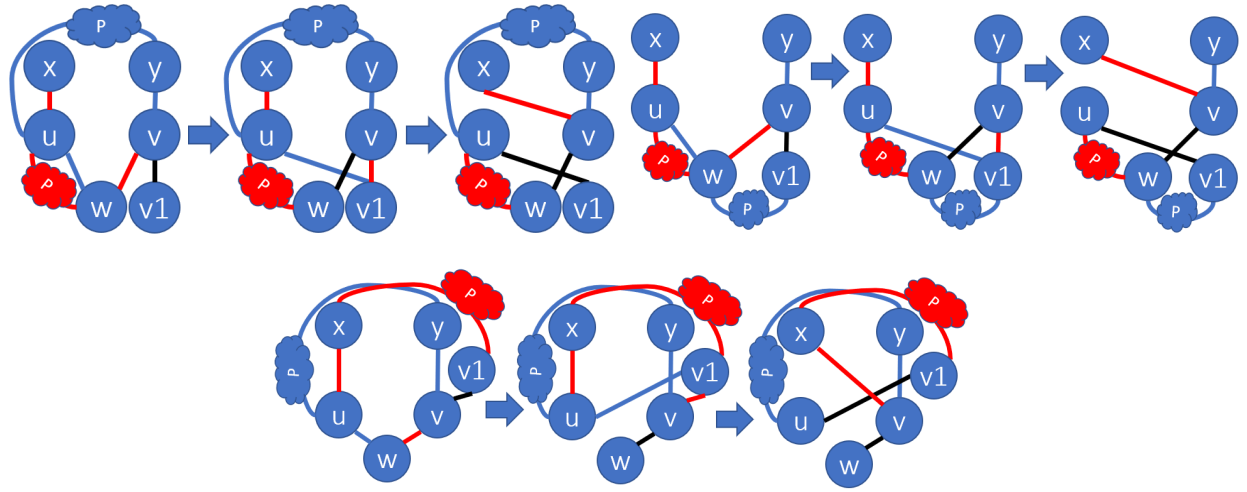


Figure 2.14: Case 2, subcase 1: (v, v_1) is black and possible configurations of blue (red) cut-paths.

since that would be handled by case 1 with an alternating 4-cycle). This swap preserves the number of components if the blue cut-path goes through $wu(p)yvv_1$, or $uw(p)v_1vy$ and in any other case we could have performed the blue double-edge swap along the cycle. In addition, this decreases k by 2 without change in the red graph. Two examples shown in Fig. 2.13.

If v has a black neighbor, v_1, and red cut-path on $xupwv_1$, blue cut-path on $wv(p)yvv_1$ or $vwpb_1vy$: we can perform a blue double-edge swap $(v, v_1), (u, w) \to (v, w), (u, v_1)$; this makes (v, w) black and (u, v_1) blue, (v, v_1) red. Now perform red double-edge swap $(v, v_1), (u, x) \to (v, x), (u, v_1)$, this makes (u, v_1) black, (v, x) red. These swaps preserve the number of connected components as shown in Fig. 2.14 top and middle.

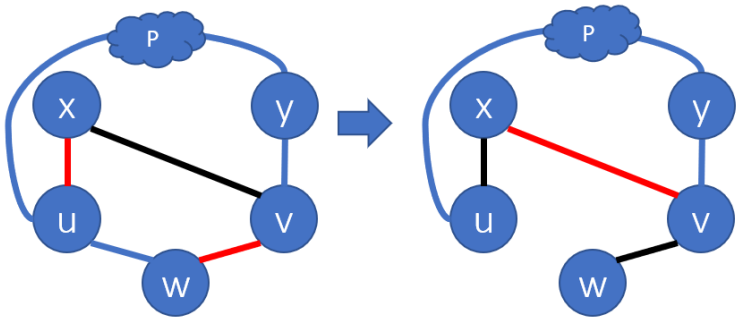


Figure 2.15: Case 2, subcase 2: blue cut-path is on $wvpyv$

If v has a black neighbor v_1, and red cut-path on $uxpv_1vw$, blue cut-path on $wv(p)yvv_1$: we can perform a blue double-edge swap $(v, v_1), (u, w) \to (v, w), (u, v_1)$; this makes (v, w) black and (u, v_1) blue, (v, v_1) red. Now perform red double-edge swap $(v, v_1), (u, x) \to (v, x), (u, v_1)$, this makes (u, v_1) black, (v, x) red. These swaps preserve the number of connected components as shown in Fig. 2.14 bottom.

Remaining subcases are symmetric from the point of view of u using its neighbors connected through red or black edges. All of these cases are maintaining number of connected components while decreasing k by 2.

Subcase 2: There is only 1 black edge: The single edge present leads to symmetric cases, here we consider the black edge present between (v, x).

If blue cut-path is on $wvpyv$, the naive double-edge swap using only blue edges would create a cycle, but using blue double-edge swap $(v, x), (u, w) \to (v, w), (u, x)$, maintain CC as shown in Fig. 2.15.

If blue cut-path is on $vwpvy$, u will have either a red or black neighbor u_1 where (v, u_1) is empty. In either case, first perform red double-edge swap $(u, u_1), (v, w) \to (u, w), (v, u_1)$, this makes (u, w) black, (v, u_1) red. If (u, u_1) was red we can stop here, otherwise (u, u_1) became blue after the swap. However, we can now perform a blue double-edge swap $(u, u_1), (v, x) \to (u, x), (v, u_1)$, this makes (u, x) and (v, u_1) black, (v, x) red. Cases shown in Fig. 2.16.

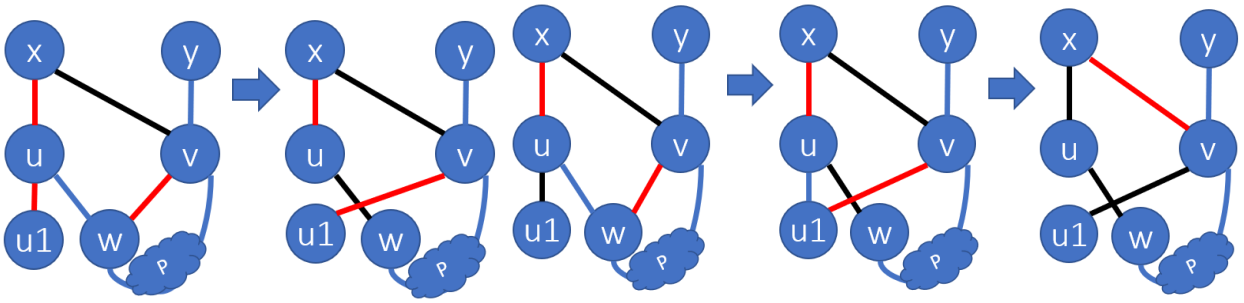


Figure 2.16: Case 2, subcase 2: blue cut-path is on $vwpvy$, (u, u_1) is red or black.

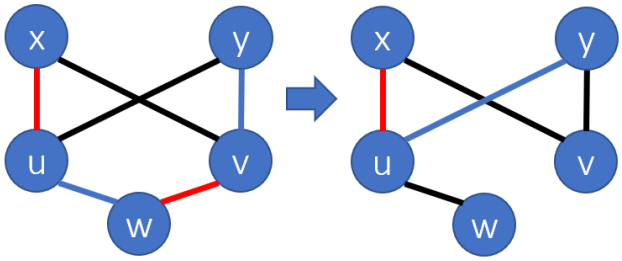


Figure 2.17: Case 2, subcase 3: 2 black edges

Subcase 3: there are 2 black edges: Double-edge swaps using pairing nodes will not affect connectivity, example shown in Fig. 2.17.

After performing any of these subcases the number of connected components will not change, but we have decreased k by 2 in every case.

Case 3: No pairing nodes exists: We can create pairing nodes without increasing the number of connected components. There will be a red, (u, x), and blue, (v, y), edge with same degree endpoints (u,v) and (x,y) in either a single large cycle (longer than 4) or 2 alternating-cycles. (u, y) and (v, x) can be only black or empty, otherwise there would be available pairing nodes. u, v will have at least one neighbor difference in both red and blue graphs.

Subcase 1: there is no black edge between endpoints: if trivial swaps increase CC, we can focus on blue cut-paths and blue connectivity and perform only blue double-edge swaps and use symmetric cases for red connectivity.

If cut-path is $uw(p)vy$, then there exists another neighbor of u not connected to v (in blue

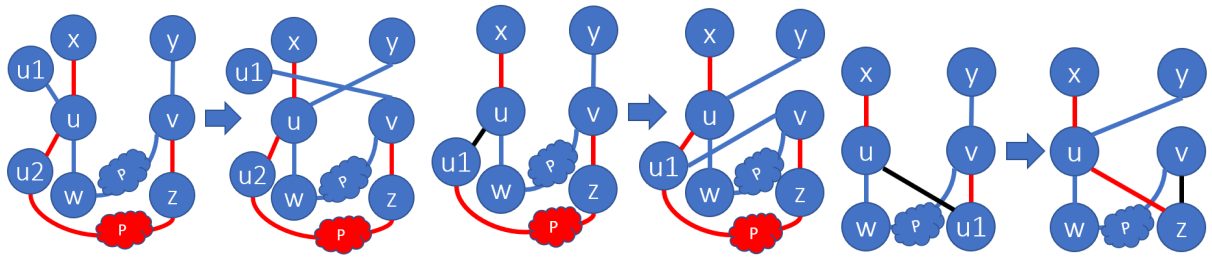


Figure 2.18: Case 3, subcase 1: possible cases when cut-path is $uw(p)vy$

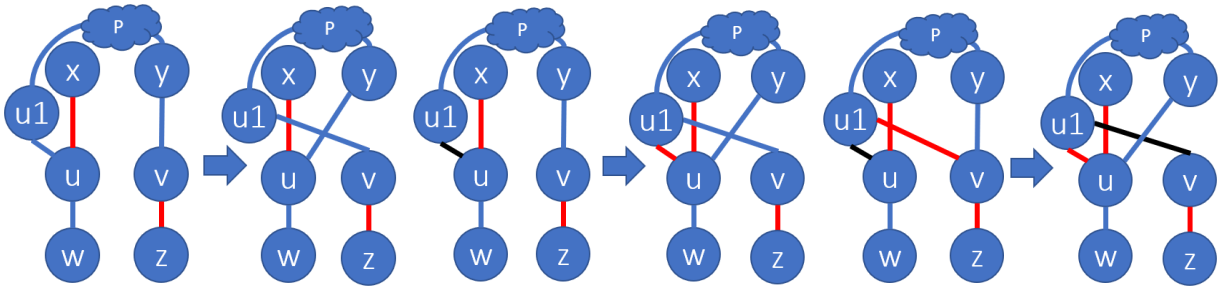


Figure 2.19: Case 3, subcase 1: possible cases when cut-path is $wuu_1(p)yv$

graph), u_1. We can perform blue double-edge swap $(u, u_1), (v, y) \rightarrow (v, u_1), (u, y)$, that makes (v, u_1) and (u, y) blue; when (u, u_1) was black it turns red, and when (v, u_1) was red it turns black. The double-edge swap creates pairing nodes x,y without change in k and other pairing nodes u,v at an increase in k by 2 when (u, u_1) was black while (v, u_1) was empty (before the swap), Fig. 2.18. If cut-path is $wuu_1(p)yv$, then the same blue double-edge swap can be performed using the first node u_1 on path, as shown in Fig. 2.19.

Subcase 2: there is a single black edge between endpoints, (v, x) (symmetric cases exists if for (u, y)): If blue edges are from different components blue double-edge swap (using blue edges only) is viable and creates pairing nodes y,x, without increasing k or number of components. (u, y) is always empty and u, v have the same neighbors in the blue graph. Now we consider if they are in the same component in blue graph and where the blue cut-paths occur:

If cut-path is $uw(p)vy$, then there exist another blue or black neighbor of u, u_1 not con-

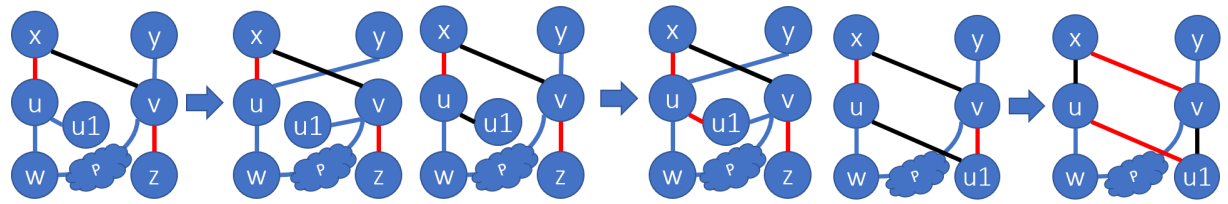


Figure 2.20: Case 3, subcase 2: possible cases when cut-path is $uw(p)vy$

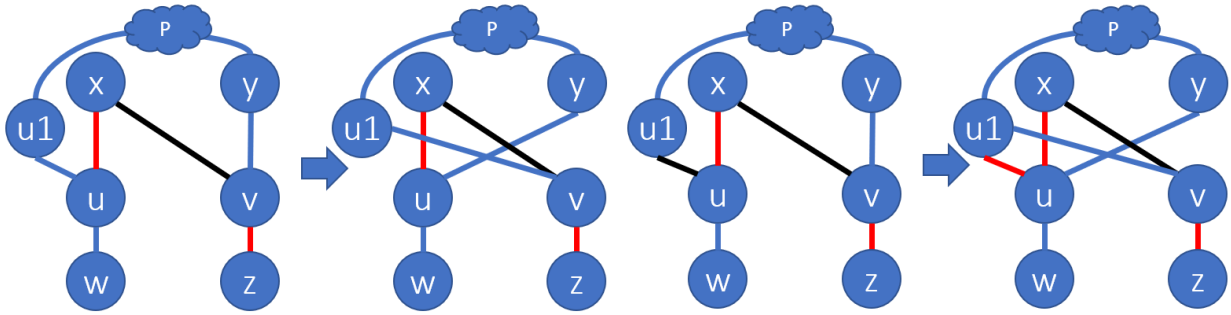


Figure 2.21: Case 3, subcase 2: possible cases when cut-path is $uu_1(p)yv$

nected to v, and not on any critical blue path. We can perform blue double-edge swap: $(u, u_1), (v, x) \to (v, u_1), (u, x)$, this makes (u, x) black, (v, x) red. If (u, u_1) was blue, then this creates pairing nodes x,y, while not increasing k. If (u, u_1) was black and (v, u_1) was empty, then this creates two pairing nodes x,y and u,v while increasing k by 2. If (v, u_1) was red, only pairing nodes are x,y and k was not increased. Of course, the number of components have not changed. Cases shown in Fig. 2.20.

If cut-path is $uu_1(p)yv$ where u_1 is not connected to v, we can perform blue double-edge swap: $(u, u_1), (v, y) \to (v, u_1), (u, y)$, that makes (u, y), (v, u_1) blue, (u, u_1) red if (u, u_1) was black initially. It creates pairing nodes, x,y without increasing k. When (u, u_1) was black, it also creates u,v pairing nodes and k increases by 2, Fig. 2.21.

If cut-path is $uu_1(p)xv$ where u_1 is not connected to v, we can perform blue double-edge swap: $(u, u_1), (v, x) \to (v, u_1), (u, x)$, that makes (u, x) black, (v, u_1) blue, (v, x) red, and (u, u_1) red if (u, u_1) was black initially. It creates pairing nodes, x,y without increasing k. When (u, u_1) was black, it also creates u,v pairing nodes and k increases by 2, Fig. 2.22.

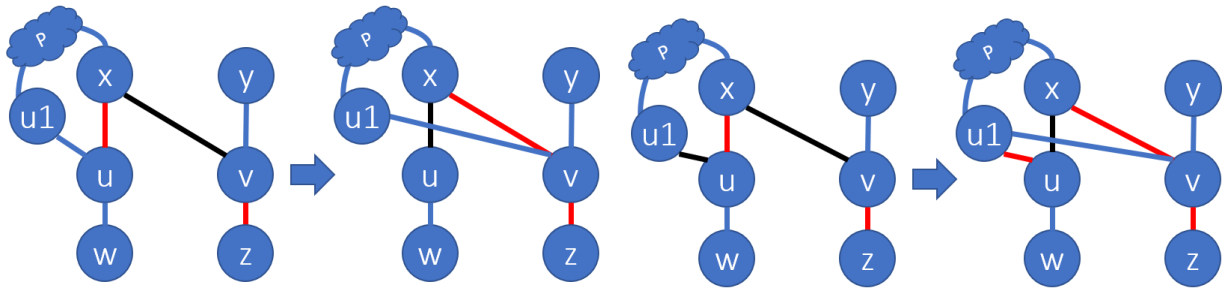


Figure 2.22: Case 3, subcase 2: possible cases when cut-path is $uu_1(p)yv$

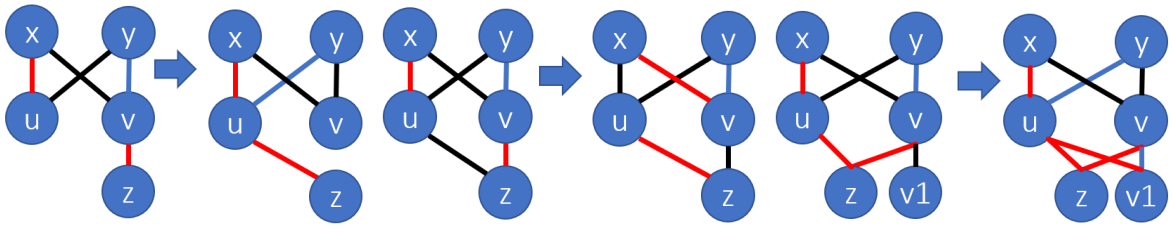


Figure 2.23: Case 3, subcase 3: possible cases depending on the color of (u, z) edge.

Subcase 3: there are two black edges between endpoints: We discuss the possible cases from the point of view of v's neighbors in the red graph, however there are symmetric cases for u as well. v has at least one red neighbor, z:

If z has no edge to u, then the red double-edge swap $(v, z), (u, y) \rightarrow (v, y), (u, z)$ makes (u, y) blue, (y, v) black, (v, z) red; and it creates pairing nodes: x,y. The swap maintains the number of components and does not increase k, Fig. 2.23 top. If z has a black edge to u, then the blue double-edge swap $(v, x), (u, z) \rightarrow (v, z), (u, x)$ makes $(v, x), (u, z)$ red and $(v, z), (u, x)$ black; and it creates pairing nodes: x,y. The swap maintains the number of components and does not increase k, Fig. 2.23 middle. Main case 2 will decrease k afterwards by 2 (4-cycles are not present since that would mean x,y were already pairing nodes). If z has a red edge to u, then there must exist a black neighbor v_1 that is not connected to u; and the red double-edge swap $(v, v_1), (u, y) \rightarrow (v, y), (u, v_1)$ makes $(u, y), (v, v_1)$ blue, (y, v) black, (u, v_1) red; and it creates pairing nodes: x,y and u,v. The swap maintains the number of components and increases k by 2, Fig. 2.23 bottom.

In all subcases for case 3, the two pairs of pairing nodes can be handled sequentially x,y first (main case 2) without changing u's or v's edges connecting to their neighbors except x, y. Then the pairing nodes, u,v can be resolved by main case 2, thus decreasing k by 2=+2-4. □

2K+MinCC+BDI: Balanced realizations for 2K+MinCC

In this section, we follow the notation from Czabarka *et al.* [16]. A graph, G, has a node partition according to their degree V_0 to $V_{d_{max}}$. V_i is the set of nodes with degree i. For every V_j, set $A_j(j) := JDM(j,j)/|V_j|$ and for $i \neq j, A_j(i) := JDM(i,j)$. Now define $\forall i, s_G(v)_i$ for every $v \in V_j$ as the number of edges from v to nodes with degree i. A realization is balanced if for all i, j pairs $s_G(v)_i \in \{\lfloor A_j(i) \rfloor, \lceil A_j(i) \rceil\}$ for all $v \in V_j$. Identically it is balanced if the floor of the difference from the average connectivity (defined by matrix A) of every node is 0, formally we define C_G the difference from balanced for a node v to a degree group i as $c_G(v,i) := \lfloor \| A_{deg(v)}(i) - s_G(v)_i \| \rfloor$; and for a degree group j as $C_G(j) = \sum_{v \in V_j} \sum_{i=1}^{dmax} c_G(v,i)$.

Balanced realizations of JDMs always exists as shown in [16]. Here we show that Lemma 4 and Corollary 5 from [16] can be applied with minor modifications to find balanced realizations of 2K with minimum number of connected components.

Lemma 2.9. *If $\exists u, v \in V_j : s_G(u)_i < \lfloor A_j(i) \rfloor < s_G(v)_i$ or $\lceil A_j(i) \rceil$ handled similarly) for a simple graph, G , then $\exists w, w' \in V_i : \{(v,w),(v,w')\} \in E, \{(u,w),(u,w')\} \notin E$ and $w, w' \neq u$; $\exists z, z' \in V_k, k \neq i$ or $\exists z \in V_k, z' \in V_{k'}, k, k' \neq i : \{(u,z),(u,z')\} \in E, \{(v,z),(v,z')\} \notin E$ and $z, z' \neq v$.*

Proof. From the initial conditions follow that $u, v \in V_i$, $deg(u) = deg(v) = i$ and $s_G(v)_i - s_G(u)_i \geq 2$. Since the sum of $s_G(v)$ equals to i and every value is an (non-negative), integer,

in worst case ($s_G(v)_i - s_G(u)_i = 2$) there $\exists k \neq i$ such that $s_G(v)_k - s_G(u)_k = 2$ or $\exists k, k' \neq i$ such that $s_G(v)_k - s_G(u)_k = 1$ and $s_G(v)_{k'} - s_G(u)_{k'} = 1$.

It follows from the previous statement, that $\exists w, w' \in V_i$ such that $\{(v,w),(v,w')\} \in E, \{(u,w),(u,w')\} \notin E$; and $\exists z, z' \in V_k$, or $\exists z \in V_k, z' \in V_{k'}$ such that $\{(u,z),(u,z')\} \in E, \{(v,z),(v,z')\} \notin E$.

We have to consider whether $w, w' \neq u$ and $z, z' \neq v$: if $w = u$, then $i = j$ and $(u,v) \in E$ ($z = v$ handled similarly). By removing (u,v) edge, the difference $s_G(v)_i - s_G(u)_i$ remains the same, which means that there exists $w, w' \neq u$. □

We just showed, that Lemma 4 in [16] has the option for both u, v to choose between two nodes for the RSO.

Theorem 2.4. *If $C_G(j) \neq 0$, there are nodes $u, v \in V_j$ and an RSO $vw, uz \to vz, uw$ transforming G into G' such that $C'_G(j) < C_G(j)$; $\forall l \neq j, C'_G(l) = C_G(l)$ and $|CC(G')| \leq |CC(G)|$.*

Proof. Now that there are at least two neighbors for both u, v two use in an RSO while applying Lemma 4 [16], we can identify cases to maintain number of connected components, based on the path between u, v and w, z:

Case 1. There is no path between u, v, *i.e.* u, v are in different connected components. Any RSO will not increase the number of connected components, thus $|CC(G')| \leq CC(G)|$.

Case 2. There is a path $v - w - p - z - u$ (where p is a simple path between w, z). An RSO using w, z would return a new path $v - z - p - w - u$ thus $|CC(G')| = CC(G)|$.

Case 3. There is a path $w - v - p - u - z$ (where p is a simple path between v, u). An RSO using w, z would return a new path $z - v - p - u - w$ thus $|CC(G')| = CC(G)|$.

Case 4. There is a path $v - w - p - u - z$ (where p is a simple path between w, u). An RSO using w, z would return new paths $v - z, w - p - u$. If there are no other paths connecting these subgraphs, then $|CC(G')| + 1 = CC(G)|$. However, using our previous observation, we can use w' that would lead to a path $w' - v - w - p - u - z$. This is in fact Case 3 using w' instead of w. □

These extensions to Lemma 4 do not change Corollary 5 in [16]; which means that application of Corollary 5 will result in the necessary sequences to return balanced realizations without increasing number of connected components. If the input G was already a MinCC realization, then the resulting G' will be both MinCC and balanced realization.

2.4 Computational hardness of dK-series

In this section, we provide the proofs for the NP-Completeness of dK-series for $d \geq 3$. We first be the definition of relevant and problems used in our reductions, then we show the NP-Completeness of 3K realizability, followed by the proofs for dK-series for $d > 3$.

2.4.1 Definitions

A $3K$-*distribution* is a distribution on graphs that can be described by two tensors: $\triangle$ and $\wedge$. $\triangle(i, j, k)$ counts the number of triples of nodes u, v, and w forming triangles such that $u \in V_i$, $v \in V_j$, and $w \in V_k$. $\wedge(i, j, k)$ counts the number of triples of nodes u, v, and w forming wedges, an induced two-path, where $u \in V_i$, $v \in V_j$, and $w \in V_k$ and v is the middle node of the two-path. The tensors have some symmetries arising from the automorphisms of the subgraphs: $\triangle(i, j, k) = \triangle(j, k, i) = \triangle(k, j, i)$ and $\wedge(i, j, k) = \wedge(k, j, i)$. The probabilities are uniform over the graphs that satisfy these constraints, i.e. have the prescribed number

of triangles and wedges between nodes of the correct degrees. These satisfying graphs are called the 3K-graphs. A natural question is: *are* there any $3K$-graphs? If there is such a graph we say it *realizes* the distribution. Formally we define the problem as:

Problem 2.2. 3K-REAL

INPUT: *a $3K$-distribution*

OUTPUT: *yes if there is a graph that realizes the $3K$-distribution, no otherwise*

Similarly, we can define realizability problems for any dK-distribution.

Our main problems used in our reductions are the following:

Problem 2.3. TRI-EDGE-PARTITION

INPUT: *a graph G*

OUTPUT: *yes if G can be edge partitioned into triangles, no otherwise*

Problem 2.4. EP_n

INPUT: *a graph G, and a positive integer n*

OUTPUT: *yes if G can be edge partitioned into K_n subgraphs (complete graphs with n nodes), no otherwise*

TRI-EDGE-PARTITION is the special case for EP_n where $n = 3$. EP_n is NP-Complete for fixed n values, where $n \geq 3$.

A 1K-distribution from a 3K-distribution

$3K$-graphs for a given $3K$-distribution have the same degree distribution and we can compute this $1K$-distribution from a given $3K$-distribution using the following equations [52]

When $k > 1$:

$$D(k) = \frac{\sum_{l} \sum_{m} \wedge(m,k,l) + \triangle(m,k,l)}{k(k-1)} \tag{2.9}$$

And for when $k = 1$:

$$D(1) = \sum_{l} \frac{2 \wedge(1,l,1) + \sum_{m \neq 1} \wedge(1,l,m) + \triangle(1,l,m)}{l-1} \tag{2.10}$$

We can view the resulting $1K$-distribution as a relaxation of the $3K$-distribution, because the $1K$-graphs are a superset of the $3K$-graphs.

2.4.2 NP-Hardness for $3K$ distributions

In this section, we will show it is NP-hard to recognize the $3K$ distributions of trees by reducing from the problem of edge partitioning a graph into triangles. Recall a $3K$ distribution is described by two tensors, $\triangle$ and $\wedge$, of dimension $d \times d \times d$ where $\triangle(i,j,k)$ counts the number of triangle induced subgraphs where one node has degree i, another has degree j, and the third has degree k and $\wedge(i,j,k)$ counts the number of induced wedge subgraphs where the first node has degree i, the middle point node has degree j, and the third node has degree k.

Given an input instance of the triangle edge partitioning problem, a graph, G, with n nodes and m edges, we will create a $3K$ distribution, 3K-Triangle-Partition$_G$, as follows. The degrees we will use in 3K-Triangle-Partition$_G$ will be 1, 4, $m/3$, and an additional degree for

each node of G, v_i, of $d_i = m/3 + 1 + i$. The two tensors for 3K-Triangle-Partition$_G$ are set as follows:

For each edge, $v_i v_j$ in G, we set:

$$\wedge(d_i, 4, d_j) = 1 \tag{2.11}$$

And for each node, v_i, we set:

$$\wedge(4, d_i, 1) = \frac{deg(v_i)}{2}(d_i - 1) \tag{2.12}$$

$$\wedge(1, d_i, 1) = \frac{deg(v_i)}{2}\binom{d_i - 1}{2} \tag{2.13}$$

$$\wedge(m/3, 4, d_i) = \frac{deg(v_i)}{2} \tag{2.14}$$

Finally we set the root constraint:

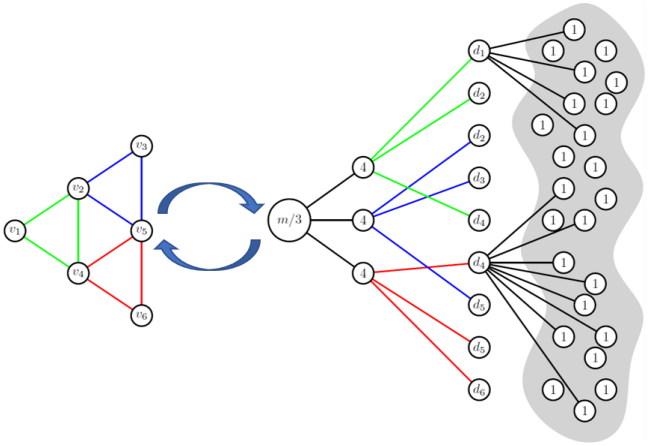
$$\wedge(4, m/3, 4) = \binom{m/3}{2} \tag{2.15}$$

All the other entries of $\wedge$ and $\triangle$ are zero.

In the following proofs, It is useful to know the $1K$-distribution computed from 3K-Triangle-Partition$_G$.

Lemma 2.10. *The $1K$ distribution of every realization of* 3K-Triangle-Partition$_G$ *is the following:* $|V_4| = m/3$, $|V_{m/3}| = 1$, $|V_{d_i}| = deg(v_i)/2$ *and* $|V_1| = m^2/3 + \sum_i \frac{deg(v_i)}{2} i$.

Proof. We can compute these values by repeatedly applying the equations from Section 2.4.1:



Edge constraints		Vertex constraints			
Λ(5,4,6)	1	**Λ(4,5,1)**	4	**Λ(4,8,1)**	14
Λ(5,4,8)	1	**Λ(1,5,1)**	6	**Λ(1,8,1)**	42
Λ(6,4,8)	1	**Λ(3,4,5)**	1	**Λ(3,4,8)**	2
Λ(6,4,7)	1	**Λ(4,6,1)**	10	**Λ(4,9,1)**	16
Λ(6,4,9)	1	**Λ(1,6,1)**	20	**Λ(1,9,1)**	56
Λ(7,4,9)	1	**Λ(3,4,6)**	2	**Λ(3,4,9)**	2
Λ(8,4,9)	1	**Λ(4,7,1)**	6	**Λ(4,10,1)**	9
Λ(8,4,10)	1	**Λ(1,7,1)**	15	**Λ(1,10,1)**	36
Λ(9,4,10)	1	**Λ(3,4,7)**	1	**Λ(3,4,10)**	1
Root constraint		**Λ(4,3,4)**	3		

Figure 2.24: On the left is a graph with a colored triangle edge partition. In the middle, there is a realization of 3K-Triangle-Partition$_G$ with the edges corresponding to the partitioning colored appropriately and most of the edges to degree one nodes omitted. On the right is the non-zero entries of 3K-Triangle-Partition$_G$ grouped into constraints for edges, nodes and the root node.

$$|V_4| = \sum_{i,j} \wedge(d_i, 4, d_j)/12 + \sum_i \wedge(m/3, 4, d_i)/6 = \sum_{e \in E(G)} 1/6 + \sum_i \frac{deg(v_i)}{12} = m/3$$

$$|V_{m/3}| = \wedge(4, m/3, 4) / \binom{m/3}{2} = 1$$

$$\begin{aligned} |V_{d_i}| &= (2\wedge(4, d_i, 1) + \wedge(1, d_i, 1)) / \binom{d_i}{2} \\ &= \left((d_i - 1)\frac{deg(v_i)}{2} + \frac{deg(v_i)}{2}\binom{d_i - 1}{2}\right) / \binom{d_i}{2} \\ &= \frac{deg(v_i)}{2}\left(d_i - 1 + \binom{d_i - 1}{2}\right) / \binom{d_i}{2} = \frac{deg(v_i)}{2} \end{aligned}$$

$$\begin{aligned} |V_1| &= \sum_i \frac{\wedge(1, d_i, 4) + 2\wedge(1, d_i, 1)}{d_i - 1} = \sum_i \frac{(d_i - 1)\frac{deg(v_i)}{2} + d(v_i)\binom{d_i - 1}{2}}{d_i - 1} \\ &= \sum_i \frac{(d_i - 1)\frac{deg(v_i)}{2} + \frac{deg(v_i)}{2}(d_i - 1)(d_i - 2)}{d_i - 1} \\ &= \sum_i \frac{deg(v_i)}{2}(d_i - 1) = \sum_i \frac{deg(v_i)}{2}(m/3 + i) = m^2/3 + \sum_i \frac{deg(v_i)}{2} i \end{aligned}$$

□

Lemma 2.11. *If G is edge partitionable into triangles, then* 3K-Triangle-Partition$_G$ *is realizable.*

Proof. Let T_1, T_2, ..., $T_{m/3}$ be an edge partitioning of G into triangles.

Create a new graph G' as follows:

- Create $m^2/3 + \sum_i \frac{deg(v_i)}{2} i$ degree one nodes, one degree $m/3$ node, and $m/3$ degree four nodes.
- For each node in G, v_i, create $deg(v_i)/2$ nodes of degree d_i and connect each of them to $d_i - 1$ of the degree one nodes.
- Connect all of the degree four nodes to the degree $m/3$ node.
- For each triangle, $T = \{(v_i, v_j), (v_j, v_k), (v_k, v_i)\}$, connect three nodes one of degree d_i, one of degree d_j, and one of degree d_k to the same node of degree four, where none of these nodes have been connected to any degree four node.

An example of this construction is shown in Figure 2.24.

Because every edge, (v_i, v_j), is in exactly one triangle, there will be two nodes one of degree d_i and one of degree d_j sharing a degree four node as a common neighbor and so $\wedge(d_i, 4, d_j) = 1$. The nodes of degree four can also be at the center of a wedge between the degree $m/3$ node and a degree d_i node. There is one wedge of this type for each node of degree d_i and so $\wedge(m/3, 4, d_i) = \frac{deg(v_i)}{2}$. Each node of degree d_i is connected to one nodes of degree four and $d_i - 1$ nodes of degree one, and so $\wedge(4, d_i, 1) = \frac{deg(v_i)}{2}(d_i - 1)$. The degree $m/3$ node has $m/3$ neighbors of degree four so $\wedge(4, m/3, 4) = \binom{m/3}{2}$. Lastly the degree d_i nodes can also be in the center of wedges with degree one node neighbors. These use up the remaining degree of each d_i node which gives $\wedge(1, d_i, 1) = \frac{deg(v_i)}{2}(d_i - 1)$.

The nodes of degree d_i can only be at the center of a wedge if the neighbors are degree four or one. All the wedges of this form have already been counted. The degree four nodes can only be at the center of the wedge if the neighbors are $m/3$ and $d - i$, but these have also been counted. Also the degree one nodes cannot be at the center of a wedge subgraph. Therefore the constructed graph realizes 3K-Triangle-Partition$_G$. □

Lemma 2.12. *If* 3K-Triangle-Partition$_G$ *is realizable, then* G *is edge partitionable into triangles.*

Proof. For each node of degree four with neighbors of degree $m/3$, d_i, d_j, and d_k, in a graph that realizes 3K-Triangle-Partition$_G$, create a set of edges $T = \{(v_i, v_j), (v_j, v_k), (v_k, v_i)\}$. These three edges exist in G because there is only a wedge between a node of degree d_i, a node of degree four, and a node of degree d_j if there is an edge in G. These sets of edges all form triangles and each edge appears in exactly one set, or otherwise we would have too many of that type of wedge. So these edge sets partition the edges of G into triangles. □

Theorem 2.5. *The problem of recognizing the* $3K$ *distributions of trees is NP-complete.*

Proof. By Lemma 2.11 and Lemma 2.12, the problem is NP-hard. It is also in NP because a graph that realizes the distribution can be verified by simply computing its $3K$ distribution. □

2.4.3 Extending to $d > 3$

We extend the proof idea from triangle edge partitioning to d-sized complete graph partitioning, the EP_n problem. This idea shows the strong relation between subgraph partitioning and the dK-series. 2K remains in polynomial time, however 3K and above is NP-Complete for the dK-series, similar to how EP_n is also only NP-Complete at $n \geq 3$.

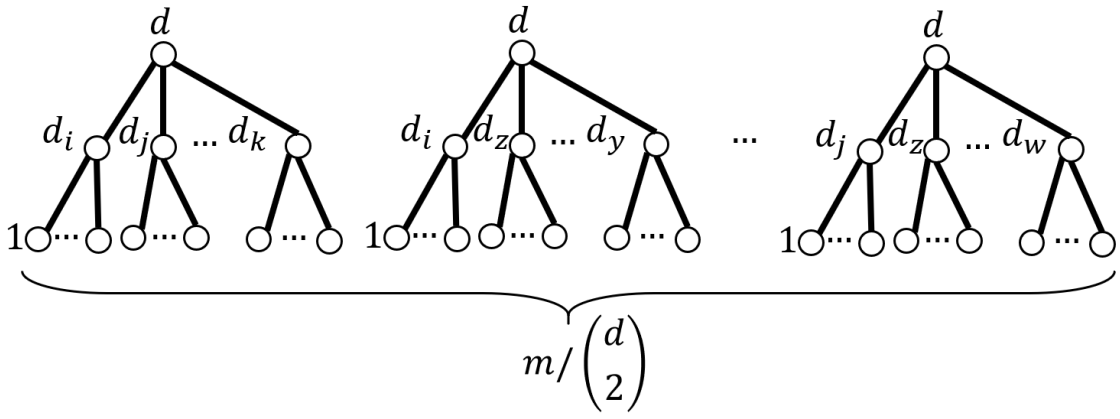


Figure 2.25: A schematic overview of the forest realization of dK-Partition$_G$ with $m/\binom{d}{2}$ trees, where every tree has a root with degree d, the first level nodes have degree d_i and there are d of them for each tree. Finally every d_i node connects to $d_i - 1$ degree-1 nodes.

Our construction follows the strategy from the 3K case as shown in Section 2.4.2 and applies it to higher d. However, the realizable instances will be forests instead of trees. This is not a fundamental difference from previous arguments, our goal with this change is to simplify the proof by minimizing the different subgraph counts that describe a forest realization instead of a tree that merges parts of the forest. We outline the modifications needed for this reduction for tree realization only in Section 2.4.3, which includes the reduction from Section 2.4.2 as a special case with $d = 3$.

From a given graph, G and d, we construct a dK-distribution, called dK-Partition$_G$. First, we denote the used degrees: 1, d and $\forall v_i \in V : d_i = d + 1 + i$.

We denote the connected d-sized degree-labeled subgraph counts of H as dK-Partition$_G(H)$, here we consider a more abstract view compared to Section 2.4.2. The assignments are possible to compute in polynomial time and space for fixed d. Figure 2.25 provides a schematic overview of the forest realization of dK-Partition$_G$, and Figure 2.26 provides the main cases of subgraph counts fit into a realization for $4K$-distribution:

First, we focus on star subgraphs centered around each d_i, there will be only two types of such graphs without or with a degree d node, we assign $\binom{d_i-1}{d-1} \cdot \frac{deg(v_i)}{d-1}$ and $\binom{d_i-1}{d-2} \cdot \frac{deg(v_i)}{d-1}$

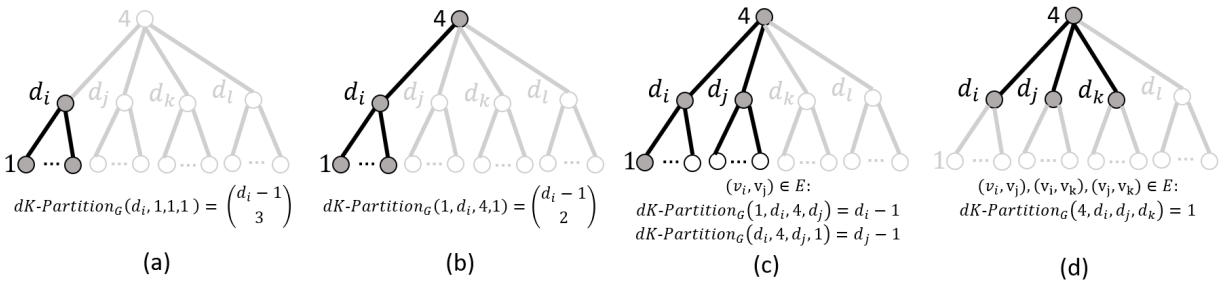


Figure 2.26: An example to assign degree-labeled subgraphs for 4K: (a) stars excluding degree-4 node for each d_i node; (b) stars including degree-4 node for each d_i node; (c) subgraphs in trees for each edge (v_i, v_j) in G: for 4K, there are only two options: using a 3-paths ending on degree-1 node connected to d_i or d_j; (d) subgraphs in trees for each triangle (v_i, v_j, v_k) in G: for 4K, degree-1 nodes are not used hence each triangle counts only once.

respectively. Figure 2.26 (a)-(b) shows an example for a single tree, however each d_i node appears in $\frac{deg(v_i)}{d-1}$ trees.

Second, we can assign proper counts for every K_p subgraph of G, where $2 \leq p \leq d-1$. Figure 2.26(c) shows an example for $4K$-distribution with $p = 2$, *i.e.* edges in G and Figure 2.26(d) shows the example for $p = 3$, *i.e.* triangles in G. It is important to notice that $p = d - 1$ complete subgraphs have only assigned as dK-Partition$_G(H) = 1$. Formally, we can describe the assignments of dK-Partition$_G(H)$ in the following way:

- We construct a rooted tree, T with node d and for each $v_i \in K_p$ add d_i nodes connected to d and finally connect sufficient number of degree-1 nodes to d_i nodes as described above.

- We have to count degree labeled subgraphs, H, of T with size d, such that each H contains d[5], every d_i nodes and all combinations of number of degree-1 nodes used from d_i nodes. The number of such configurations of H is a combination with replacement to choose $d-p-1$ degree-1 neighbors for each d_i nodes (total p). Following the notation of [**?**], this is $(\binom{p}{d-p-1}) = \binom{d-p-1+(p-1)}{d-p-1} = \binom{d-2}{d-p-1}$, which captures the number of

[5]node d has only degree p during this construction, but we replace that p with value d for dK-Partition$_G(H)$.

different configurations of H, but not the assignment for dK-Partition$_G(H)$.

- We can assign dK-Partition$_G(H) = \prod_i \binom{d_i-1}{f(H,d_i)}$ where $f(H, d_i)$ returns the number of degree-1 nodes attached to d_i in H, and i is the index of nodes in K_p.

Finally, we set every other entry in dK-Partition$_G$ to 0.

The number of nodes with for different degrees are the following: $|V_1| = \sum_i \frac{deg(v_i)}{d-1} \cdot (d_i - 1)$, $|V_d| = m/\binom{d}{2}$, $|V_{d_i}| = \frac{deg(v_i)}{d-1}$. Although, these values can be calculated in principle for any fixed d, we have not found a simple closed form description. This is due to the different possible configurations of H and the normalization required by these configuration. The most straightforward way to compute $1K$-distribution, is to compute $3K$-distributionfrom dK-Partition$_G$ while being careful about double-counting wedges for dK-Partition$_G$ and use equations show in Section 2.4.1. However, even this approach requires the enumerate all possible configurations of H.

Note that the size of dK-Partition$_G$ is polynomial in $|V|$, since the number of complete subgraphs is $O(|V|^p)$ for each $2 \leq p \leq d-1$ in $G(V, E)$ and the different configurations of subgraphs of each tree T is $O((d-2)^{d-3})$ where d is a fixed constant ($d \geq 3$) and independent of the size of G. The order of polynomial and the constants quickly grow for higher values of d, however, we consider fixed constant d values as in the dK-seriesand it would be considered in increasing order (1K, 2K, 3K, 4K, 5K, etc.) rather than an arbitrary large value of d such as $d = 200$.

Lemma 2.13. *If G can be edge partitioned into K_d subgraphs, then* dK-Partition$_G$ *is realizable.*

Proof. Given a valid edge partition of G into K_d subgraphs $\{H_1, ..., H_{m/\binom{d}{2}}\}$ we can construction a realization of dK-Partition$_G$:

- Create $m/\binom{d}{2}$ number of degree d nodes; $\forall v_i \in V$, $\frac{deg(v_i)}{d-1}$ number of degree d_i nodes.
- For each node of degree d_i, we connect them to $d_i - 1$ degree 1 nodes.
- For each H_j and for every endpoint $v_i \in H_j$, we connect d_i nodes (that have not been connected to a degree d node) to the same d degree node.

By construction, the degrees of these nodes as they are intended to be and matching of dK-Partition$_G$. If we consider the counts of d-sized connected subgraphs, we can easily verify that the constructed graph has the correct dK-Partition$_G$.

□

Lemma 2.14. *If* dK-Partition$_G$ *is realizable, then* G *can be edge partitioned into* K_d *subgraphs.*

Proof. Given a realization of dK-Partition$_G$, we can assign the edge partitioning for G into K_d subgraphs in the following way: every degree d node connects to exactly d nodes with d_i degrees. These nodes can be matched back to nodes in G by their degrees, furthermore these nodes are forming a complete subgraph of size d in G. The edges of these K_d subgraphs are only used once, since otherwise the realization would contain more than the prescribed number of d-sized subgraphs from dK-Partition$_G$, hence the proposed edge partitioning is valid.

□

Theorem 2.6. *The realizability problem of the dK-distributionof forests or realizability of a dK-distributionis NP-complete for any constant* $d \geq 3$.

Proof. Based on Lemmas 2.13 and 2.14, dK-Partition$_G$ has a realization if and only if the EP_d instance is a yes instance. Because any realization can be verified in polynomial time by

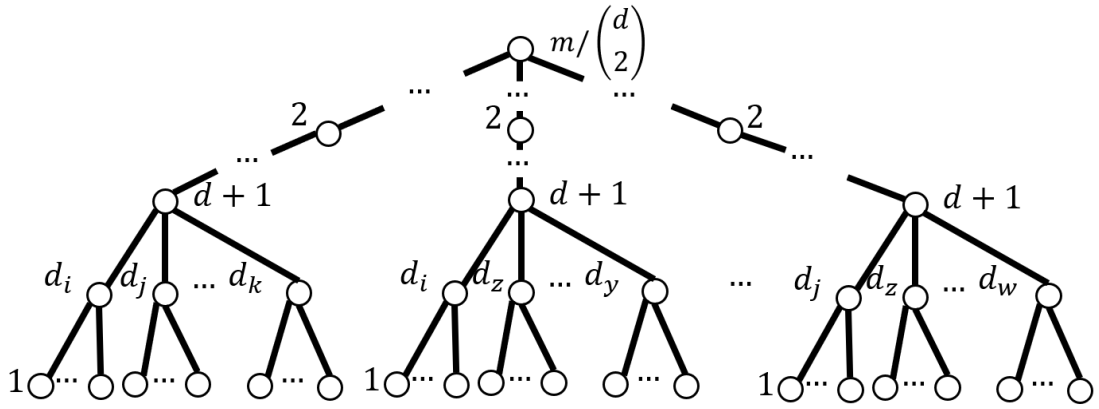


Figure 2.27: A schematic overview of the tree realization of dK-Partition$_G$, it has a root with degree $m/\binom{d}{2}$, and degree two nodes to separate subtrees with roots of degree $d+1$ (previously d).

counting d-sized connected subgraphs, the realizability of a dK-distributionis NP-complete.

□

Tree Realizations

It is possible to extend the construction of dK-Partition$_G$, such that realizable instances always form a tree instead of a forest with $m/\binom{d}{2}$ trees. We follow the idea from Section 2.4.2 and extend dK-Partition$_G$ with a single node of degree $m/\binom{d}{2}$; the previously degree d nodes will have degree $d+1$; and finally we add $c = \lfloor \frac{d-3}{2} \rfloor \cdot m/\binom{d}{2}$ degree 2 nodes.

We use the node with degree $m/\binom{d}{2}$ to the merge the independent trees of the forest realization into a single tree. We use degree-2 nodes to separate the subtrees using simple paths of $\lfloor \frac{d-3}{2} \rfloor$ nodes between degree $m/\binom{d}{2}$ and degree $d+1$ nodes. This ensures that no information is shared by d-sized subgraphs in dK-Partition$_G$ between two subtrees rooted at degree $d+1$ nodes. A schematic overview of this construction is shown in Figure 2.27.

The assignments of H configurations that include nodes with these new degrees has to be added to dK-Partition$_G$. However, these counts follow previous arguments and it is possible to mechanically define them based on the previous construction of dK-Partition$_G$.

Table 2.1: Real-life topologies used for evaluation.

Dataset	$\|V\|$	$\|E\|$	Avg Deg.	$\bar{c}$	# dorms	# years
FB: Rice [69]	4 087	184 828	90.45	0.294	10	22
FB: Princeton [69]	6 596	293 320	88.94	0.237	57	27
FB: UCSD [69]	14 948	443 221	59.30	0.227	40	23
FB:New Orl. [72]	63 392	816 884	25.77	0.222	-	-
amazon0601 [49]	403 364	2 443 309	12.11	0.42	-	-
youtube-links[55]	1 134 894	2 987 623	5.26	0.081	-	-

The number of degree-1 nodes ($\sum_i \frac{deg(v_i)}{d-1} \cdot (d_i - 1)$) has to be also adjusted along with $d_i = max(d, m/\binom{d}{2}) + 1 + i$ for each $v_i \in V$; this is to ensure that d_i degrees are greater than both d and $m/\binom{d}{2}$.

2.5 Simulations for Real-World Undirected Graphs

In this section, we perform simulations targeting properties of real-world undirected graphs, and we evaluate the performance of different construction algorithms in practice.[6] This is also a use-case of our work: people often need to produce topologies that resemble graphs like the online social networks listed in Table 2.1. The main finding of this evaluation is that our construction algorithms can target 2K+clustering well, and orders of magnitude faster than prior MCMC state-of-the-art: reducing the time from days and weeks (or not even terminating for large graphs, to minutes and tens of seconds.

Datasets

We evaluate all proposed algorithms in terms of accuracy and efficiency on a variety of real-world topologies. Table 2.1 summarizes the topologies, which are divided in two groups.

[6]The algorithms were implemented in Python using NetworkX [40] and executed on an AMD Opteron 2.4Ghz machine. A C++ implementation would be potentially faster by a constant factor especially, if combined with a more recent, faster CPU. We also contributed our software to NetworkX [32]. However, the focus in this section, is the comparison of different algorithms.

Those in the first group are relatively smaller (*i.e.* up to $15K$ nodes) Facebook university networks (Rice, Orinceton, UCSD) that come with several annotated node attributes. The second group consists of larger graphs (*i.e.* more than $60K$ nodes) without node attributes (FB: New Orleans, amazon0601, youtube-links).

First, we compute the properties (JDM, clustering, etc) of the *original* real topology (last line in each topology), then we use different construction algorithms to target those properties (listed on the lines above). For each topology, we construct 20 realizations with different algorithms: `2K_Simple`, `2K+S` with two different values of the clustering parameter S ($S = 1$ and another S selected to target $\bar{c}$), and `2K_Simple_Attributes` where node attributes are "dorms" and "year of admission". Table 2.2 reports the properties of the original and constructed graphs, including properties explicitly targeted ($\bar{c}$ and the degree assortativity for dorms, and year) and non-targeted ones (average shortest path length, average closeness, number of maximal cliques), averaged over realizations.

The last two columns report the time (in sec) to construct the graph. The second column from the end ("Construction") refers to the time it takes for the construction algorithm to terminate. The last column ("MCMC") refers to the *additional time* that our improved MCMC would use, starting from the realization the construction algorithm produced, to further target degree-dependent clustering within $NMAE < 2\%$.

Properties

Here we examine whether targeting either $\bar{c}$ or attributes, in addition to JDM, brings the constructed graphs closer to the original w.r.t. other non-targeted properties as well. For the first three small graphs, we observe that the average shortest path and average node closeness of graphs produced by `2K_Simple` is already close to the original graph. Thus, targeting $\bar{c}$ does not match better the non-targeted properties or the assortativity of node attributes,

Table 2.2: Graphs are constructed targeting different properties of 6 different original topologies. Graph properties are averaged over 20 runs. Last two columns report the time for the Construction algorithms and for MCMC to target $\bar{c}(k)$.

Topology	Graph	Graph properties						Constr. Time (sec)	MCMC Time (sec)
		Avg Node Value			Number of Cliques	Assortativity			
		$\bar{c}$	Sh.P.	Closn.		Dorms	Year		
FB Rice	2K Simple	0.06	2.33	0.43	425K	0.01	0.01	2.52	9091
	2K+S=1.0	0.53	2.74	0.37	11.5M	0.01	0.01	20	193
	2K+S=0.52	0.29	2.68	0.38	3.1M	0.01	0.01	21	249
	2K+dorm	0.13	2.38	0.43	793K	0.42	0.09	3.45	773
	2K+year	0.09	2.36	0.43	500K	0.08	0.28	3.43	2249
	original	0.29	2.44	0.41	1.1M	0.42	0.28	-	-
FB Princeton	**2K Simple**	0.04	2.49	0.40	530K	0.00	0.01	3.69	16K
	2K+S=1.0	**0.55**	2.97	0.34	29.0M	0.00	0.01	29	**274**
	2K+S=0.57	**0.24**	2.97	0.34	9.8M	0.00	0.01	**27**	**331**
	2K+dorm	0.10	2.56	0.40	751K	0.09	0.27	5.70	2324
	2K+year	0.08	2.59	0.39	755K	0.05	0.45	5.32	2157
	original	**0.24**	2.67	0.38	1.3M	0.09	0.45	-	-
FB UCSD	2K Simple	0.01	2.86	0.35	438K	0.00	0.01	4.90	66K
	2K+S=1.0	0.63	3.39	0.30	3.7M	0.00	0.01	46	920
	2K+S=0.61	0.23	3.46	0.29	5.4M	0.00	0.01	43	1656
	2K+dorm	0.03	2.88	0.35	526K	0.25	0.05	8.45	30K
	2K+year	0.02	2.89	0.35	476K	0.02	0.36	7.52	42K
	original	0.23	2.98	0.34	743K	0.25	0.36	-	-
FB: New Orl.	2K Simple	0.00	3.89	0.26	760K	-	-	18.65	524K
	2K+S=1.0	0.58	4.46	0.23	1.5M	-	-	79	3150
	2K+S=0.74	0.30	4.56	0.22	2.4M	-	-	74	9360
	original	0.22	4.35	0.24	1.5M	-	-	-	-
amazon0601	2K Simple	0.00	4.84	0.21	2.4M	-	-	53	∞
	2K+S=1.0	0.61	6.04	0.17	637K	-	-	239	∞
	2K+S=0.73	0.42	5.92	0.17	1.1M	-	-	214	∞
	original	0.42	6.39	0.16	1.0M	-	-	-	-
youtube-links	2K Simple	0.00	4.64	0.22	2.9M	-	-	71	∞
	2K+S=0.69	0.14	5.07	0.18	2.4M	-	-	955	∞
	2K+S=1.0	0.21	4.82	0.20	2.2M	-	-	1073	∞
	original	0.08	5.34	0.19	3.3M	-	-	-	-

which stays close to zero. However, targeting a given attribute significantly improves $\bar{c}$ and the assortativity of the second attribute (*e.g.* $2K{+}dorms$ in Princeton) in addition to exactly achieving assortativity for "Dorms", also improves $\bar{c}$ from 0.04 to 0.08 and assortativity for

the attribute "Year" from 0.01 to 0.27 when compared to `2K_Simple`.

In the three larger graphs, targeting a higher $\bar{c}$ than what is achieved by `2K_Simple` brings the average path length and closeness significantly closer to the original graph. For example, in the *amazon0601* topology `2K_Simple` achieves an average node closeness of 0.21, whereas $2K + S = 0.73$ achieves 0.17 which is closer to the real value of 0.16. Finally, for all graphs, the property "Number of Cliques" does not consistently improve by targeting either $\bar{c}$ or attributes.

Efficiency

The time needed to generate a graph using either of our three construction algorithms is similar, which is expected since they all run in linear time in $|E|$. For example, it takes tens of seconds to generate synthetic graphs for all Facebook topologies of Table 2.1 even when we target maximum clustering (*i.e.* $2K+S = 1.0$). As a baseline for comparison, we also targeted $2K + \bar{c}$ (*i.e.* 2.25K) with MCMC using double edge swaps, which is the previous state-of-the-art. With naive MCMC, it took approximately *a day* to generate synthetic graphs with the target $\bar{c}$ for the smallest topologies (Rice and Princeton); while simulations for the bigger graphs did not finish after several days.

Recall that the last column of Table 2.2 reports the time that a 2K-preserving MCMC needs to target the degree-dependent clustering of the original graph (2.5K), starting from the realization constructed by our (2K or 2.25K) construction algorithm. We observe that the time for the MCMC to target 2.5K increases as we decrease the average clustering in the generated graphs. We also observe that the larger the graph, the worse the MCMC matches the graph. For example, in the largest examples (Amazon, Youtube graphs), our improved MCMC did not successfully target 2.5K in these cases. One reason behind is that the cost of local updates and number of swaps is large for large graphs.

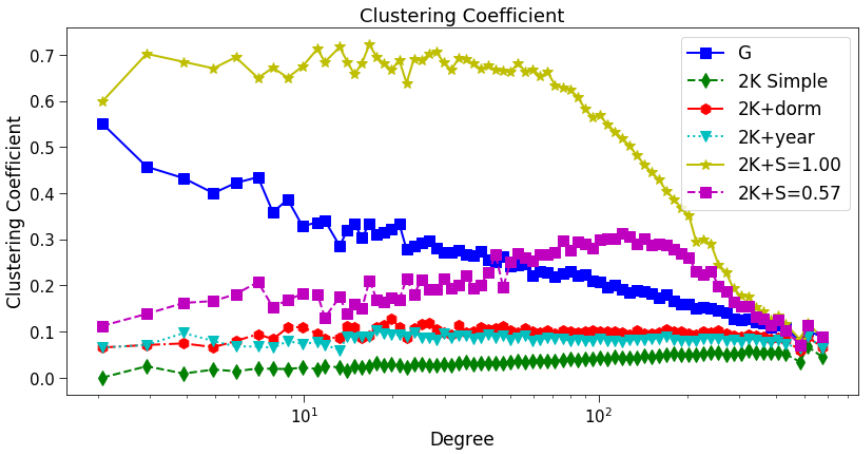


Figure 2.28: Average degree-dependent clustering coefficient for the FB Princeton graph. Figure shows $\bar{c}(k)$ for the original graph, G, and for a realization produced by `2K_Simple`, 2K+year, 2K+dorm, $2K+S=0.57$ and $2K+S=1$.

Finally, we observe that construction targeting maximum average clustering (*i.e.* $2K+S=1$) has a faster MCMC than 2.25K construction, if the end goal is to follow up construction with MCMC to target 2.5K. Figure 2.28 further elaborates on this point by zooming in on the FB Princeton graph: the blue graph is the real 2.5K ($\bar{c}(k)$) of the original graph; all other curves plot the $\bar{c}(k)$ achieved by all other construction methods. `2K_Simple`, 2K+year, 2K+dorm achieve low clustering ($\bar{c}=0.04, 0.1, 0.08$), much lower than the original graph ($\bar{c}=0.24$). $2K+S=0.57$ matches average clustering ($\bar{c}=0.24$) but not $\bar{c}(k)$. $2K+S=1$ significantly overshoots the real $\bar{c}(k)$ in most degree groups. These realizations serve as starting point for the 2.5K-targeting MCMC, at the end of which $\bar{c}(k)$ is within $NMAE<2\%$ of the target $\bar{c}(k)$. The yellow graph on top shows the $\bar{c}(k)$ at the end of $2K+S=1$ (*i.e.* maximum clustering); it turns out that starting from there and using MCMC is the fastest (274 sec) in Table 2.2), because it is easier to destroy rather than create triangles with MCMC. The purple graph corresponds to 2.25K ($2K+S=0.57$), which does not match the degree-dependent clustering compared to the original graph and takes longer to fix with MCMC (331sec) in Table 2.2.

Discussion

The benefits of our approach, compared to prior MCMC approaches are two-fold: (i) accuracy, *i.e.* how well we match 2K (exactly), and average clustering $\bar{c}$ (approximately) and (ii) construction running time. As we can see in Table 2.2, both approaches and MCMC can achieve close to $\bar{c}$ (column 2), if allowed to run long enough. However, as it can be seen in the last two columns of Table II, our running time is on the order of seconds or up to tens of seconds, while MCMC running time varies from 100s of seconds (minutes) up to hundreds of thousands of seconds (several weeks); in the cases of the larger graphs (amazon, youtube), the MCMC approach does not even converge to the target (thus ∞ time). The magnitude of the reduction of running time depends on the graph characteristics, and is amplified when the target graphs (i) are large (*i.e.* large $|V|, |E|$), (ii) exhibit high clustering (*e.g.* see original $\bar{c}$ in Table 2.2), and (iii) are sparse (as indicated by their average degree in Table 2.1).

The Facebook university graphs all have almost the same $\bar{c}$ and are ordered in increasing size and sparsity in Table 2.1: Rice, Princeton, UCSD, New Orleans. In Table 2.2, we can see that the corresponding difference in running time is in the same order, *i.e.* amplified with size and sparsity. The amazon dataset is an order of magnitude larger and sparser but has a higher target average clustering than the Facebook networks. In this case, the MCMC never converges (indicated by ∞ time in the last column of Table 2.2), while our algorithms still terminate on the order of minutes. The reason is that it is highly unlikely to create triangles by chance (MCMC or pure 2K), compared to our more structured 2K+ construction (where we create as many triangles as we can using $2K+S=1$), then we destroy triangles using an improved MCMC). Fig. 2.28 shows an example of how sortedness was used to overshoot degree-dependent clustering before applying MCMC. Therefore, targeting sparser graphs with higher clustering is more challenging for the MCMC approach, while 2K+S was significantly faster. Sparse (pairs of) degree groups tend to have low clustering

if we only consider 2K (not 2K+S). We have not experimented with datasets where the graph is dense and the target clustering is low (but realizable); our intuition is that even 2K construction would achieve close to target clustering in that case, since it tends to generate graphs with low clustering. Finally, sparsity can affect the running time of our algorithm in practice (asymptotically it is still $O(|E| \cdot d_{max})$) in a different way: sparse graphs might require fewer NeighborSwitches (the most expensive operation in our algorithm) compared to dense graphs.

2.6 Summary

In this chapter, we have considered 2K and additional constraints. We have provided an efficient way to construct simple undricted graphs (`2K_Simple`), that exhibit exactly a target degree correlations and potentially additional properties, including: clustering (2.25K and 2.5K), number of connected components (2K+CC), node attributes (2K+A). We have developed efficient heuristics (2K+S) and MCMC approaches for 2.25K and 2.5K and we showed that the realizability problem for such distributions are NP-Complete. Furthermore, we have shown that the realizability for dK-distributions for any fixed $d \geq 3$ are also NP-Complete.

Our results pushed the boundary of what was previously possible to target efficiently within the dK-series from just degree sequences and degree correlations to additional properties. This is necessary in order to target real-world graphs such as online social networks. We have shown that including triangles (or clustering coefficients) lead to NP-Complete problems and provided efficient heuristics. However, we were able to show positive results on the number of connected components.

Chapter 3

Directed Graph Construction

3.1 Introduction

In this chapter, we consider directed graphs in general, and directed acyclic graphs (DAGs) as a special case. To the best of our knowledge, we are the first to define and target directed degree correlations, which we refer to as directed 2K (D2K) [67]. In our main approach, D2K, we represent directed graphs as bipartite graphs with non-chords and we target the bipartite JDAM to construct simple directed graphs. In our second approach, we further restrict the notion of directed degree correlation to D2.1K, to capture in-, out-degree correlations for a partition of nodes into the same degree groups (nodes with the same in and out degree); we show that D2.1K can be solved targeting JDAM with a more granular partitioning of attributes. In addition to the above contributions, which first appeared in [67], in this chapter we also (i) provide a heuristic to target the number of mutual edges in a directed graph and (ii) we show that D2K and D2.1K always have BDI realizations, similarly to the undirected case of 2K. For the special case of DAG construction, we develop : (i) a flow-based approach for DAG1K construction (*i.e.* DAGs with a target degree sequence) that is more efficient

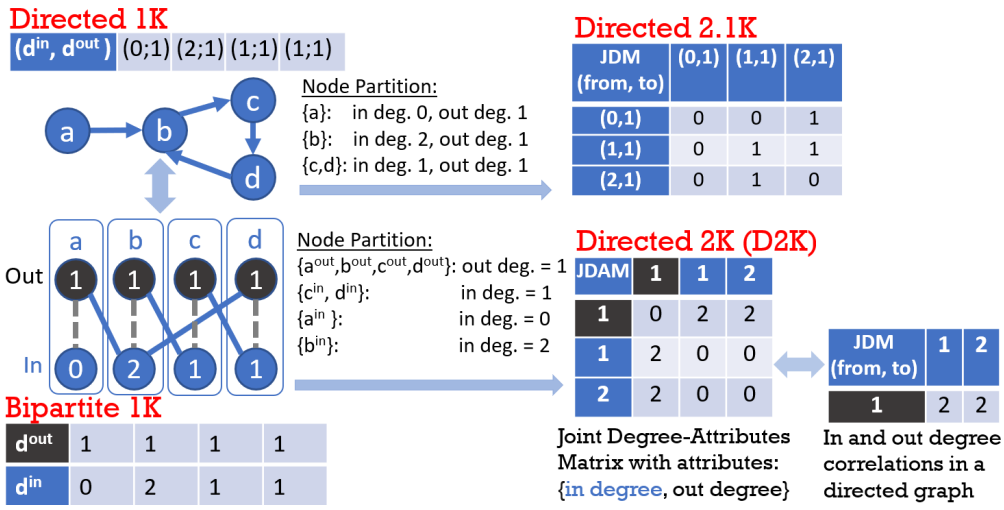


Figure 3.1: Defining Directed 2K, to capture degree correlations in a directed graph: top left, Directed 1K; bottom left, Bipartite 1K with non-chords (shown in dashed line); bottom right, Directed 2K (D2K); top right: Directed 2.1K.

than previous methods; and (ii) we solve a special case of DAG2K, namely level graphs.

The outline of this chapter is as follows. Section 3.2 describes the core idea of D2K construction including realizability conditions, an efficient construction algorithm and an importance sampling algorithm. Section 3.3 describes extensions of D2K such as D2.1K, number of mutual edges and balanced realizations. Section 3.4 evaluates the convergence of directed dK-series on real-world graphs. Section 3.5 defines the extension of dK-series to DAGs. Section 3.5.1 describes a simple network flow-based approach to improve efficiency of realizability and construction of DAG1K graphs. Section 3.5.2 describes the current state of the possible approaches to solve DAG2K and we show a solution for a special case of level graphs (D2K+L). Section 3.6 evaluates the convergence of DAGdK-series on real-world graphs. Section 3.7 summarizes this chapter.

3.2 Directed 2K Construction

Our goal in this chapter is to go beyond just directed degree sequence and capture directed degree correlation. One approach would be to simply consider the degree correlations between in and out degrees in a directed graph, as shown in Fig. 3.1-bottom rightmost matrix. Alternatively it is possible to work with the equivalent representation of a directed graph as an undirected bipartite graph without non-chords (Fig. 3.1-bottom left), and define degree correlations there. We partition in and out nodes by their degree, essentially considering that nodes in the bipartite graph can have an attribute that takes two values, "in" or "out." We can now define degree correlation using the Joint Degree-Attribute Matrix (JDAM), as shown on Fig. 3.1-bottom right. This leads to a JDAM with two attribute values, such that $\forall k, l = 1, ..., d_{max}$ degrees and $i \in \{in, out\}$ attribute values $JDAM(\{k, i\}, \{l, i\}) = 0$, *i.e.,* because the bipartite graph has no edges between two "in" or two "out" nodes. Furthermore, the number of non-chords will be noted as $f(\{k, i\}, \{l, j\})$, where $k, l \in \{1, ..., d_{max}\}$ and $i \neq j \in \{in, out\}$; f can be computed by passing through the directed degree sequence once and counting the number of entries with in-degree k and out-degree l.

We note that this notion of *Bipartite JDAM* is a special case of $JDAM$ and inherits all the properties known for $JDAM$. It allows us to get rid of the directionality of the edges and handle a regular undirected JDAM using the 2K+A algorithm previously defined. For D2K, the main challenge is to show that the non-chords described by the directed degree sequence can be avoided. In summary we define the D2K problem as follows, and an example is shown on Fig. 3.1.

D2K. The input is two target properties, namely the JDAM$^{\odot}(\{k, i\}, \{l, j\})$ with two attribute values (in and out) and the directed degree sequence DDS$^{\odot}$. The goal is to construct a simple directed 2K-graph with these target properties (construction) if such realizations exist (realizability).

3.2.1 Realizability

Recall that in our D2K definition, nodes are partitioned into at most $2d_{max}$ parts $V_{\{k,in\}}$, $V_{\{k,out\}}, k = 1, ..., d_{max}$, according to the distinct combinations of degrees and attributes they exhibit and JDAM$(\{k,i\},\{l,j\})$ is indexed accordingly. For example, on Fig. 3.1 bottom-right, each node belongs to one of four parts $V_{\{0,in\}} = \{v \in V : d^{in} = 0\}$, $V_{\{1,out\}} = \{v \in V : d^{out} = 1\}$, $V_{\{1,in\}} = \{v \in V : d^{in} = 1\}$, $V_{\{2,in\}} = \{v \in V : d^{in} = 2\}$, and the JDAM is 3x3 (by removing rows and columns corresponding to any $V_{\{0,i\}}$, since there are no edges using these parts of any partition).

We define the necessary and sufficient conditions for a target D2K, *i.e.,* JDAM$^{\odot}(\{k,i\},\{l,j\})$ and DDS$^{\odot}$, to be realizable as the following:

I $\forall k,l,i : JDAM(\{k,i\},\{l,i\}) = 0$

II $\forall k,l,i,j$, if $JDAM(\{k,i\},\{l,j\}) > 0$, $JDAM(\{k,i\},\{l,j\}) + f(\{k,i\},\{l,j\}) \leq |V_{\{k,i\}}| \cdot |V_{\{l,j\}}|$

III $\forall k,i : |V_{\{k,i\}}| = \sum_{\{l,j\}} \frac{JDAM(\{k,i\},\{l,j\})}{k}$ = number of times k appears in DDS as i.

These are generalizations of the conditions for an undirected JDM, JDAM to be realizable, and they are clearly necessary. The first condition states that every realization of the target JDAM is bipartite, *i.e.* there should be no edges between two nodes both in "in" or "out" parts. The second condition considers edges between two ("in" and "out") parts and states that the number of edges defined by the $JDAM(\{k,i\},\{l,j\})$ plus the number of non-chords (shown as $f(\{k,i\},\{l,j\})$) should not exceed the total number of edges possible in a complete bipartite graph across the two parts. The last condition ensures that the target JDAM and the target DDS are consistent: the number of nodes with in (or out) degree k should be the same whether computed using the JDAM or the DDS. The conditions are shown to be

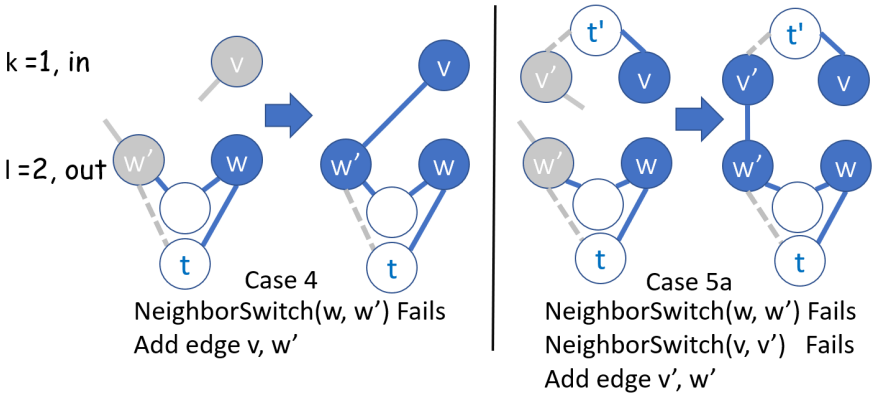


Figure 3.2: New cases in Algorithm 3.1, while attempting to add (v, w) edge.

Algorithm 3.1 `D2K_Simple`

Input: $DDS^{\odot}$, $JDAM^{\odot}$
Initialization:
a: Create G with nodes, partition, stubs using $DDS^{\odot}$
b: Add non-chords to G using $DDS^{\odot}$
Add Edges:
1: **for** $(\{k,i\},\{l,j\}) \in JDAM^{\odot}(\{k,i\},\{l,j\})$
2: **while** $JDAM(\{k,i\},\{l,j\}) < JDAM^{\odot}(\{k,i\},\{l,j\})$
3: Pick any nodes $v \in V_{\{k,i\}}$, $w \in V_{\{l,j\}}$
s.t. (v, w) is not a non-chord or existing edge
4: **if** v does not have free stubs:
5: v': node in $V_{\{k,i\}}$ with free stubs
6: NeighborSwitch(v,v')
7: **if** NeighborSwitch fails, $v := v'$
8: **if** w does not have free stubs:
9: w': node in $V_{\{l,j\}}$ with free stubs
10: NeighborSwitch(w,w')
11: **if** NeighborSwitch fails, $w := w'$
12: add edge between (v, w)
13: $JDAM(\{k,i\},\{l,j\})$++; $JDAM(\{l,j\},\{k,i\})$++
Output: directed graph representation of G

sufficient via the constructive proof of the algorithm. Necessity of these conditions for simple graph construction are trivial.

3.2.2 Algorithm for D2K Construction

First, we create a set of nodes V, where $|V| = 2|\text{DDS}|$, we assign stubs, non-chords to each node and partition nodes, as specified in the *target directed degree sequence* $\text{DDS}^{\odot}$. We also initialize all entries of JDAM to 0.

Then the algorithm proceeds by connecting two nodes (one from "in" and one from "out" side, because of Condition I, thus adding one edge (v, w) at a time, that (i) do not form an edge (ii) do not have a non-chord between them (to avoid self-loops in the directed graph representation) and (iii) for whom the corresponding entry in the JDAM has not reached its target. The added complexity from JDAM construction lies in the non-chords and the fact that NeighborSwitch operation can "fail". This failure means that there is no suitable node to perform NeighborSwitch with due to a non-chord constraint. However, in these cases another edge can be added as shown in Fig. 3.2. Next, we prove that this is indeed always the case.

Proof. Condition I ensures that every realization is bipartite, Condition II guarantees that two nodes can be always chosen to add an edge following the arguments in Lemma 2.1 and Condition III ensures that at least one node exist with a free stub in every part of the partition if $JDAM(\{k, i\}, \{l, j\}) < JDAM^{\odot}(\{k, i\}, \{l, j\})$ using Lemma 2.2. Now, we show that every iteration can proceed by adding a new edge to the graph. The cases are identical to `2K_Simple` as long as NeighborSwitch operation can be executed without using non-chords. This leads to two additional cases:

Case 4. Add a new edge between a node w w/out free stubs and a node v w/free stubs (or w/out free stubs where NeighborSwitch is possible) where NeighborSwitch is not possible for w using w' without using any non-chords. In this case w' has the same neighbors as w except the one for which it has an assigned non-chord. In this case w' is not connected to v

and it is possible to add $\{w', v\}$ edge ($\{w', v\}$ is clearly not an edge since then v would be also connected to w or w could have done a NeighborSwitch).

Case 5. Add a new edge between two nodes (v, w) w/out free stubs, where neither can do a NeighborSwitch with v' and w' respectively. We break this case into two subcases, based on whether nodes v', w' (with free stubs) form a non-chord.

Case 5a. v', w' is not a non-chord. This means that we can add a new edge between v', w'. It is easy to see that v', w' edge is not already present, because otherwise v and w could have performed a NeighborSwitch.

Case 5b. v', w' is a non-chord. This case is not possible when v, w are not able to perform NeighborSwitches at the same time. Without loss of generality, let's say that v connects to every neighbor of v' and w'. This means that no NeighborSwitch is available for v. Now, if we want to construct w such that it can't perform a NeighborSwitch with w', w would connect to every neighbor of w'; however, this would include v too, and clearly that edge doesn't exist. Contradiction.

This concludes our proof and shows that the algorithm will terminate and generate a bipartite graph after adding $|E|$ edges without using non-chords. □

Running Time. Since the algorithm is essentially the same as before, the running time is $O(|E| \cdot d_{max})$. The only difference is when a NeighborSwitch fails to free up stubs, we use a node with free stubs. However, this takes only a constant operation when compared to `2K_Simple`. The final directed graph can be constructed from the bipartite representation by collapsing nodes with non-chords and assigning directions to edges appropriately, this takes $O(|V| + |E|)$ time.

Space Complexity. The `D2K_Simple` algorithm requires an additional $O(|V|)$ space compared to `2K_Simple` to store non-chords. However, the overall space complexity remains

unchanged: $O(|V| + |E|)$.

3.2.3 Space of realizations

The algorithm for the directed case has the same properties for generating any realization as `2K_Simple`. In this section we focus on double-edge swaps for MCMC-based sampling.

D2K is a special case of an undirected JDAM, and thus inherits the property that JDAM realizations are connected via 2K-preserving double-edge swaps [16],[4] if non-chords are allowed (equivalently, self-loops in directed graphs). However, we cannot use the known swaps to sample from the space of simple directed graphs.

The space of simple realizations of directed degree sequences (D1K) is connected over double edge swaps, that preserve (in and out) degrees, and triangular C_6 swaps. If the difference between two realization is the orientation of a directed three-cycle, then the triangular C_6 swap consists of edge rewirings such that the orientation of the cycle is reversed in a single step. The sufficiency of only these two types of swap was shown in [25]. The necessity of these swaps also carries over to (simple) directed 2K realizations. However, Fig. 3.3 shows a counterexample (a directed 4-cycle) where the classic swaps are not sufficient to transform one realization to the other, thus requiring a more complex swap. We leave it as an open question whether tight upper bounds can be derived on the swap size for Directed 2K realizations.

There are possibly other cases where swaps must be more complex and include more edges at once, for example larger directed cycles with specific in/out degree order. In this dissertation, we do not provide tight upper bounds on the number of self-loops (in the directed graph representation) or the size of swaps required, but we do emphasize that no multi-edges are required and the number of self-loops are of course bounded by $|V|$.

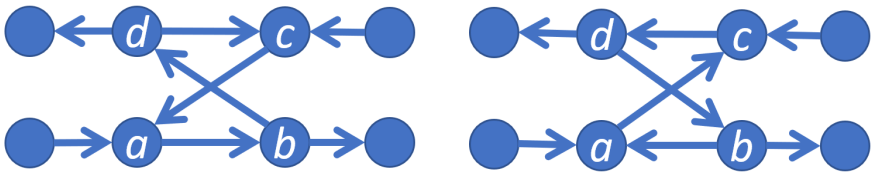


Figure 3.3: Realizations of the same D2K input without JDAM-preserving double-edge (or C_6) swaps that would not use any self-loops. The edges along the directed 4-cycle must change their direction simultaneously.

3.2.4 Importance sampling

In the previous section, we have showed that MCMCs are not possible using the "usual" double-edge swaps or any small fixed sized swaps (e.g, like triangular C_6 swaps used for D1K). However, we will shortly show how to extend Bassler *et al.* [9] results to work with D2K inputs. The key observation is already present in Lemma 2.9, that we will reuse in Section 3.3 to construct balanced realizations of D2K inputs. If we can show that there are always two options to pick a neighbor to perform some local switch, we can avoid the non-chords during the algorithms.

Bassler's paper [9] is built on the idea that after partial assignments from node to degree groups, the rest of a bipartition between degree group k-l admits a balanced realization. In Theorem 2. and 3. in [9], we can easily apply the previous observation about two options, since in these theorems when a local rewiring happens the difference in degrees is always at least two. With these observations, it is possible to sample Degree Spectra Matrices (matrix with $|V| \times d_{max}$, entry describes number of edges from node to degree group) correpsonding to D2K inputs. The rest of the arguments would be verbatim from [9] and essentially follow as before: the degree spectra matrices will decompose to bipartite degree sequence problems (with single non-chords per nodes) per degree group pairs. The importance sampling from these bipartite sequences are well understood [12, 18] and the union of these bipartite sequences return realizations to the original D2K input as in [9].

The possibility to sample from degree spectra matrices of D2K instances tells us something about the space of realizations too. It shows us that space of D2K realizations have "islands" where double-edge swaps and triangular C_6 swaps are sufficient to connect any pair of realizations and these "islands" are the union of realizations of degree spectra matrices where two degree spectra matrices are neighbors through double-edge (or triangular C_6) swap. However, this degree spectra matrix sampling still doesn't specify the maximum swap size that is required to get from D2K realizations to another, *i.e.* our previous claim about larger required swaps for the space to be connected still holds.

3.3 D2K with additional constraints

We show three examples that capture more information of graphs by imposing more restrictions on the realizations. The first approach, D2.1K, fixes average in (and out) degree neighborhoods in directed graphs; the second approach, D2K+M, is a simple heuristic to achieve high number of mutual edges in realizations, the third approach considers Balanced Degree Invariant D2K realizations. Other properties can be considered as well, similarly to the undirected graph construction.

D2.1K: correlation between (in, out) degree pairs. Instead of working with the bipartite representation, we can work directly with the directed graph, as in Fig. 3.1-top right. We partition nodes by both their in and out degrees (d_v^{in}, d_v^{out}), and we can define the joint degree matrix to capture the number of edges $JDM((k^{in}, l^{out}), (m^{in}, n^{out}))$, between nodes with (k^{in}, l^{out}) and (m^{in}, n^{out}) degrees.

D2.1K is a natural extension of the undirected 2K and captures a more restrictive notion of degree correlation than our main D2K definition. We use the notation D2.1K, since it already contains the information for a corresponding D2K. D2.1K fixes the average degree

neighborhoods, since for a given node, v, (with known in degree) D2.1K describes the in degrees of nodes that connect to v (similarly to out degrees as well). This is not specified in D2K, since it doesn't consider the in and out degree at the same time. However, we can also observe that D2.1K can be transformed into a D2K instance with additional attributes. D2K with additional attributes (D2K+A) is the same kind of generalization used to get from JDM to JDAM, and our results from D2K carry over to D2K+A. If we use the additional attribute to capture the nodes' in and out degree, then the resulting D2K+A instance is equivalent to D2.1K. Therefore a simple extension of `D2K_Simple` can solve D2.1K instances.

D2K+M: Number of Mutual Edges. This work was motivated by the observation that, in sparse graphs, D2K produced an order of magnitude less mutual (reciprocated) edges than in the original social networks. We use a heuristic approach to target number of mutual edges in a directed graph during construction, by greedily adding mutual edges when permitted by degree and JDAM constraints. In `D2K_Simple` line 12-13, we can check if the non-chord pairs of v, w can form an edge and add it if possible. We denote this approach as D2K+M or D2.1K+M following the notation from UMAN where "M" represents the number of mutual edges in a graph. This heuristic works well in practice as shown in Section 3.4, but exact solutions might be difficult to achieve.

D2K+BDI: Balanced Realizations. We can find a swap sequence from any D2K realization to a balanced realization for D2K graphs using our observation from Section 2.3.4. Since there will be two nodes to pick from at every double-edge swap when applying Lemma 4 from [16], it is possible to avoid self-loops while transforming a D2K realization to a D2K+BDI. Since every node has one non-chord assigned, we can simply pick a node for the double-edge swap that does not from a non-chord.

Lemma 3.1. *If $C_G(j) \neq 0$, then there are nodes $u, v \in V_j$ and an RSO $vw, uz \to vz, uw$ transforming G into G' such that $C'_G(j) < C_G(j)$ and $\forall l \neq j, C'_G(l) = C_G(l)$ if every node participates in exactly one non-chord.*

Table 3.1: Input graphs from SNAP [49]

Name	#Nodes	#Edges	Generation (sec)
p2p-Gnutella08	6,301	20,777	0.474
Wiki-Vote	7,115	103,689	1.894
AS-Caida	26,475	57,582	2.066
Twitter	81,306	1,768,135	44.884

Proof. As shown in Lemma 2.9, there are at least two neighbors for both u, v two use in an RSO while applying Lemma 4 [16]. We can find an RSO without using non-chords, since u has exactly one non-chord, it can pick (at least) one of w, w' and similarly v can pick one of z, z' for an RSO. □

As before, we can apply Lemma 4 and Corollary 5 from [16] in combination with the previous lemma to produce a balanced realizations for any realizable D2K inputs.

3.4 Simulations for Real-World Directed Graphs

We have the same simulation setup as in Section 2.5. We compare realizations generated by Directed ER (D0K), UMAN, Directed Degree Sequence (D1K), Directed 2K, Directed 2K+M, Directed 2.1K, Directed 2.1K+M with the corresponding target properties captured on input graph (G). Since we are the first to introduce D2K, we focus on how well D2K targets various graph properties, rather on evaluating the algorithm efficiency.

We used examples of directed graphs from SNAP [49]: p2p-Gnutella08, Wiki-Vote, AS-Caida (without customer relations), Twitter. Table 3.1 provides an overview of the graphs (without self-loops and multi-edges) used in our experiments and reports the average time to construct realizations using D2K for these examples. In the rest of this section, we consider several well known graph properties as listed below with brief descriptions where appropriate. A subset of these properties were also used by Orsini et. al. [57] to study the convergence of dK-series

for undirected networks and we report those that are more natural for directed graphs, such as the triad census. We average results over 20 realizations for every construction method and specific property.

1. *Dyad Census* counts the different configurations for every pair of nodes: "mutual" - edges in both direction, "asymmetric" - edge only in one direction and "null" - no edge present.

2. *Triad Census* counts the non-isomorphic configuration for every triplet of nodes. A complete list of configurations and naming conventions can be found in [43]. Configurations are identified by three numbers (mutual, asymmetric, and null counts) and a letter in case of different non-isomorphic configuration with the same number of edges. For example "003" is a triplet of nodes where none of the edges are present, "030C" is a directed 3-cycle and "300" is a triplet of nodes where all directed edges are present.

3. *Dyad-wise Shared Partners* for pairs of nodes can be defined in three ways for directed graphs: using independent two-paths, using shared outgoing neighbors or using shared incoming neighbors between pairs of nodes [62]. Dyad-wise shared partners (DSP) count node pairs by the number of shared partners appearing in a network.

4. *Average Neighbor Degree* captures the average degree of a nodes' neighbors, and we split this property for in - and out degrees. Similarly, we refer to *Expansion* for directed graphs as the ratio of the first hop and second hop neighborhoods' sizes going out, or coming in to a node. These properties capture similar aspects of a network, but expansion excludes any mutual edges or edges between nodes in the first hop neighbors.

5. *Betweenness Centrality CDF*, *Shortest Path Distribution*, *K-Core Distribution*, *Eigenvalues*

Results. The size of these graphs matches the inputs by definition. We can also observe

in Fig. 3.4 that Directed Degree Distributions and Degree Correlations are captured by D2K, D2.1K as expected by definition. On the other hand, D0K, D1K and UMAN capture Degree Correlations poorly, thus D2K graphs have a chance to capture other properties more accurately than D0K or D1K.

Dyad Census is not well captured for Twitter, as we can see in Fig. 3.4. However, there are order of magnitude improvements in the number of mutual edges between D2.1K (123,040.4) and D2K (3,628.7), D1K (2,155.95) or D0K (233.05). Of course, UMAN preserves this property by definition. D2K+M and D2.1K +M does not meet the target exactly since the current implementation is a heuristic, but it significantly boosts the number of mutual edges.

Triad Census is surprisingly well captured by UMAN, the reason being the exact match for the Dyad Census in the previous point. On the other hand, a convergence can be seen between dK-series generators with significant improvements in dense triadic structures from D1K to D2K and from D2K to D2.1K. Targeting the mutual edges helps D2K and D2.1K in the dense triadic structures like “201”, “210” or “300”.

Dyad-wise Shared Partners follow similar trends to other properties, such that D2.1K is significantly more accurate than D2K. D2K improves over D1K in terms of ”outgoing shared partners” but that improvement decreases at ”independent two-paths” and disappears at ”incoming shared partners”.

Expansion is again best approximated by D2.1K and D2.1K even matches *Average Neighbor Degree* exactly if marginalized by degrees as in Fig. 3.4. D2K also follows the general shape of these distributions but includes larger error, while D1K has systematic difference compared to G.

Betweenness Centrality CDF has no significant improvements after matching degree distributions with D1K in Twitter; other examples reached target closer with D1K. Interestingly UMAN performs almost identically to D0K, even though the number of mutual edges is

Table 3.2: Summary of results: showing improvements by fixing more properties. Labels: "." - no improvement, "-" - decreased accuracy, "+" - increased accuracy, "Exact" - matched by definition.

Property	UMAN→D1K	D1K→D2K	D2K→D2.1K
Degree Distribution	Exact	Exact	Exact
Degree Correlation	+	Exact	Exact
Dyad Census	-	+	+
Triad Census	+	+	+
Betweenness Centrality	+	.	.
Shortest Path Distribution	+	+	+
Eigenvalues	+	+	+
DSP	+	+	+
Expansion	+	+	+
Avg. Neighbor degrees	+	+	Exact
S. Connected Components	.	.	.
K-Core Distribution	+	.	+

significantly different.

Shortest Path Distribution has slow convergence to target across different methods, but the average shortest path is shorter than the observed in G.

K-Core Distribution is best captured by D2.1K, and there is a small improvement from D1K to D2K using Twitter. However, the dense core using D1K or D2K is almost an order of magnitude lower core index. Targeting mutual edges for D2K helps in reconstructing better structure in terms of coreness, and gets the results closer to D2.1K.

Eigenvalues of Twitter is again best targeted by D2.1K. There is a difference between leading eigenvalues in graph realizations of the other methods but starting at the second eigenvalue the difference between D1K and D2K quickly decreases. D2K+M shows significantly lower error than D2K.

Table 3.2 gives an overview of how network properties are affected by the different dK graph construction methods for the other remaining networks, results are presented in Figures 3.5-3.7. The Twitter network showcased most of our general findings, but individually some

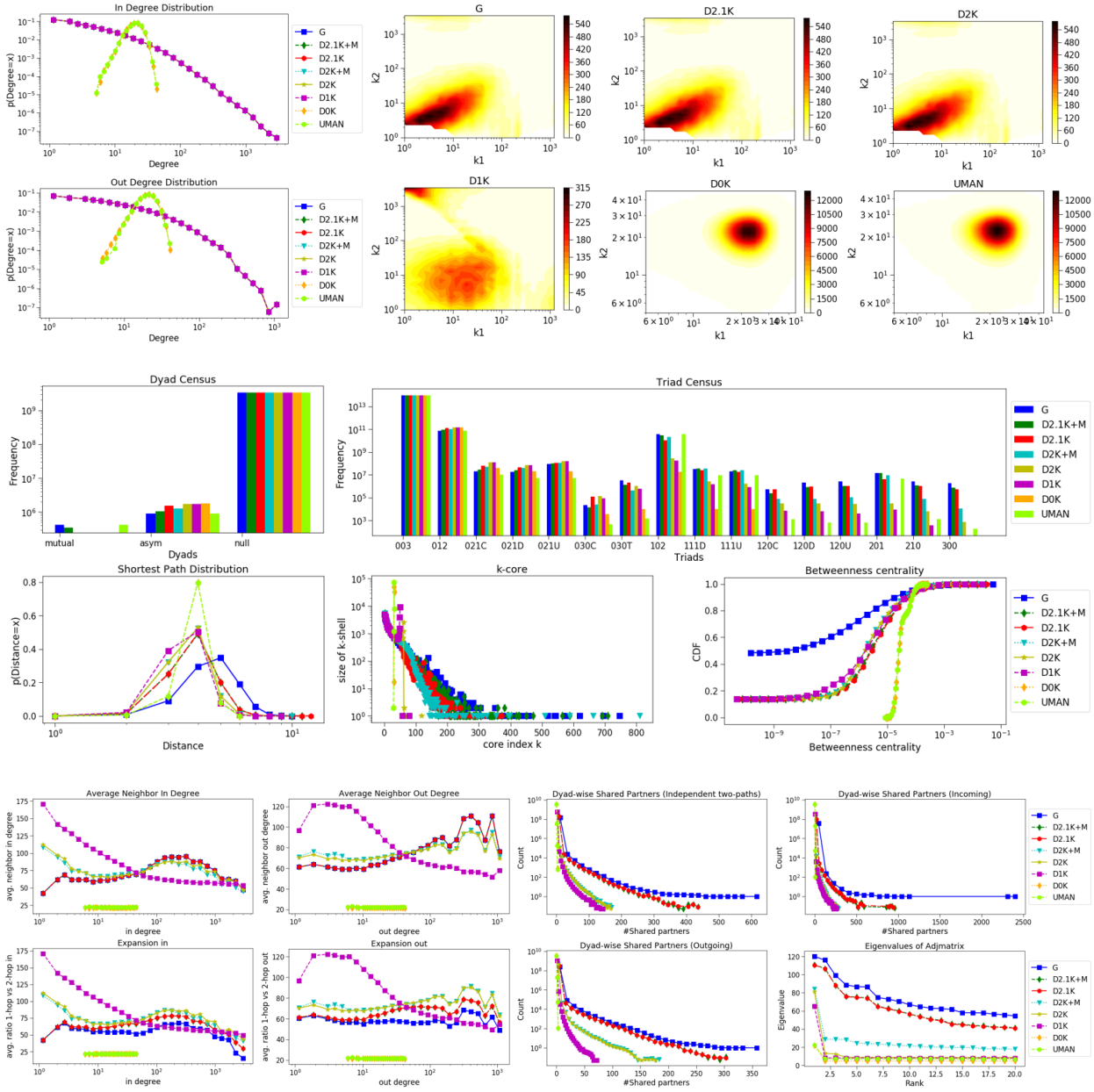


Figure 3.4: Results for Twitter graph: Directed Degree Distribution, Degree Correlation, Dyad-, Triad Census, Shortest Path Distribution, K-core distribution, Betweenness Centrality, Expansion, Average Neighbor Degree, DSP and top 20 Eigenvalues

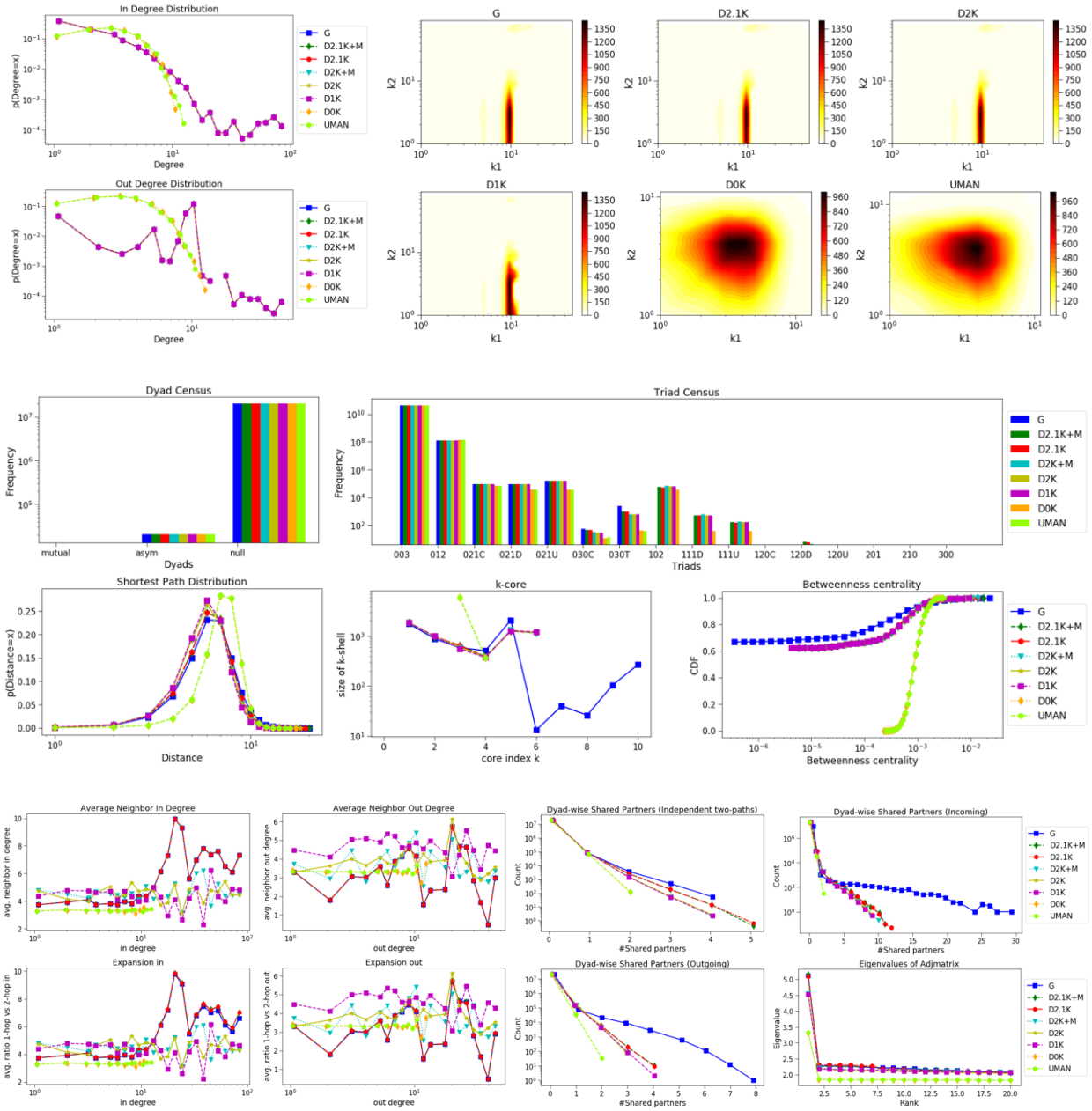


Figure 3.5: Results for p2p-Gnutella08 graph: Directed Degree Distribution, Degree Correlation, Dyad-, Triad Census, Shortest Path Distribution, K-core distribution, Betweenness Centrality, Expansion, Average Neighbor Degree, DSP and top 20 Eigenvalues

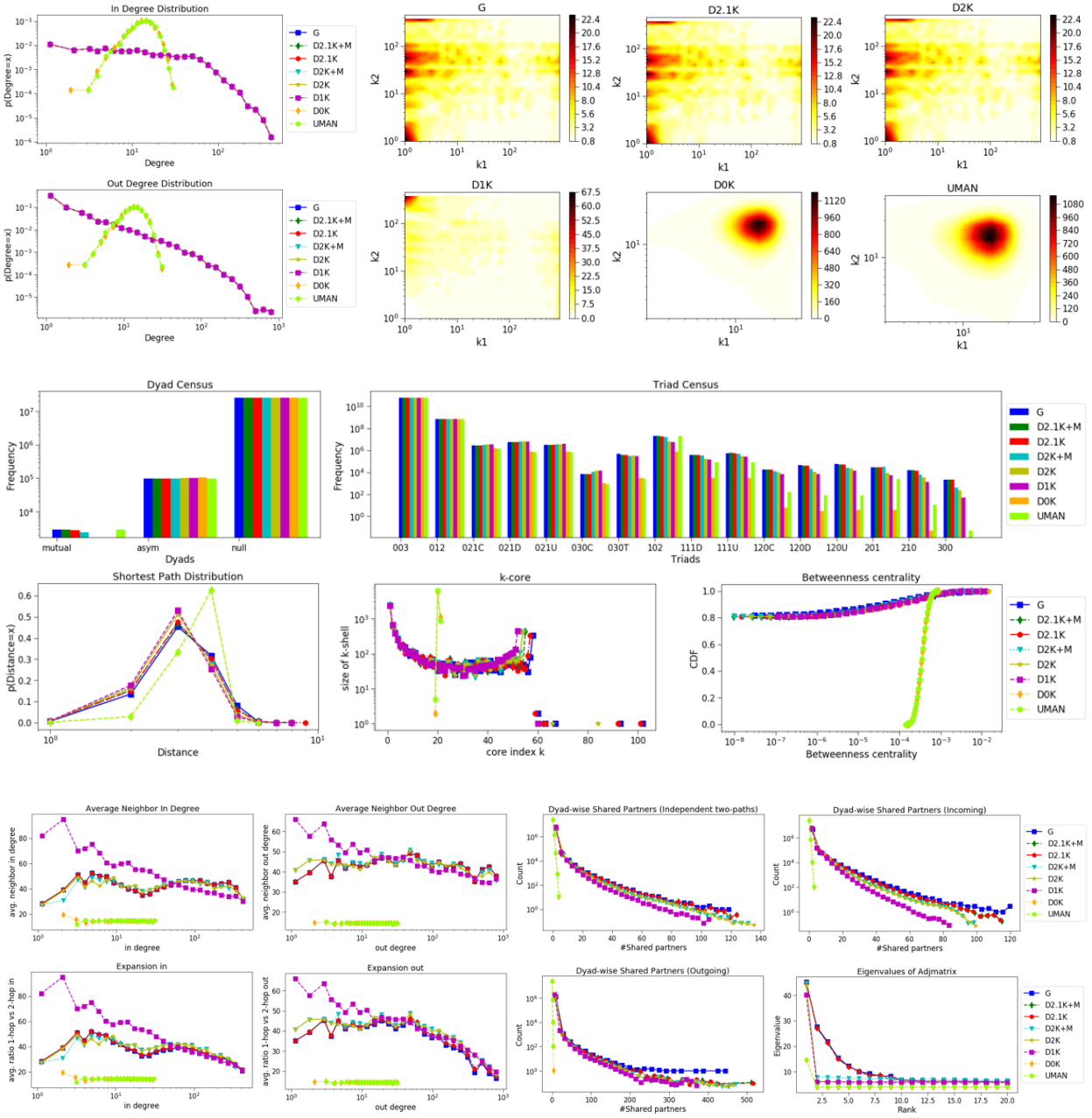


Figure 3.6: Results for Wiki-Vote graph: Directed Degree Distribution, Degree Correlation, Dyad-, Triad Census, Shortest Path Distribution, K-core distribution, Betweenness Centrality, Expansion, Average Neighbor Degree, DSP and top 20 Eigenvalues

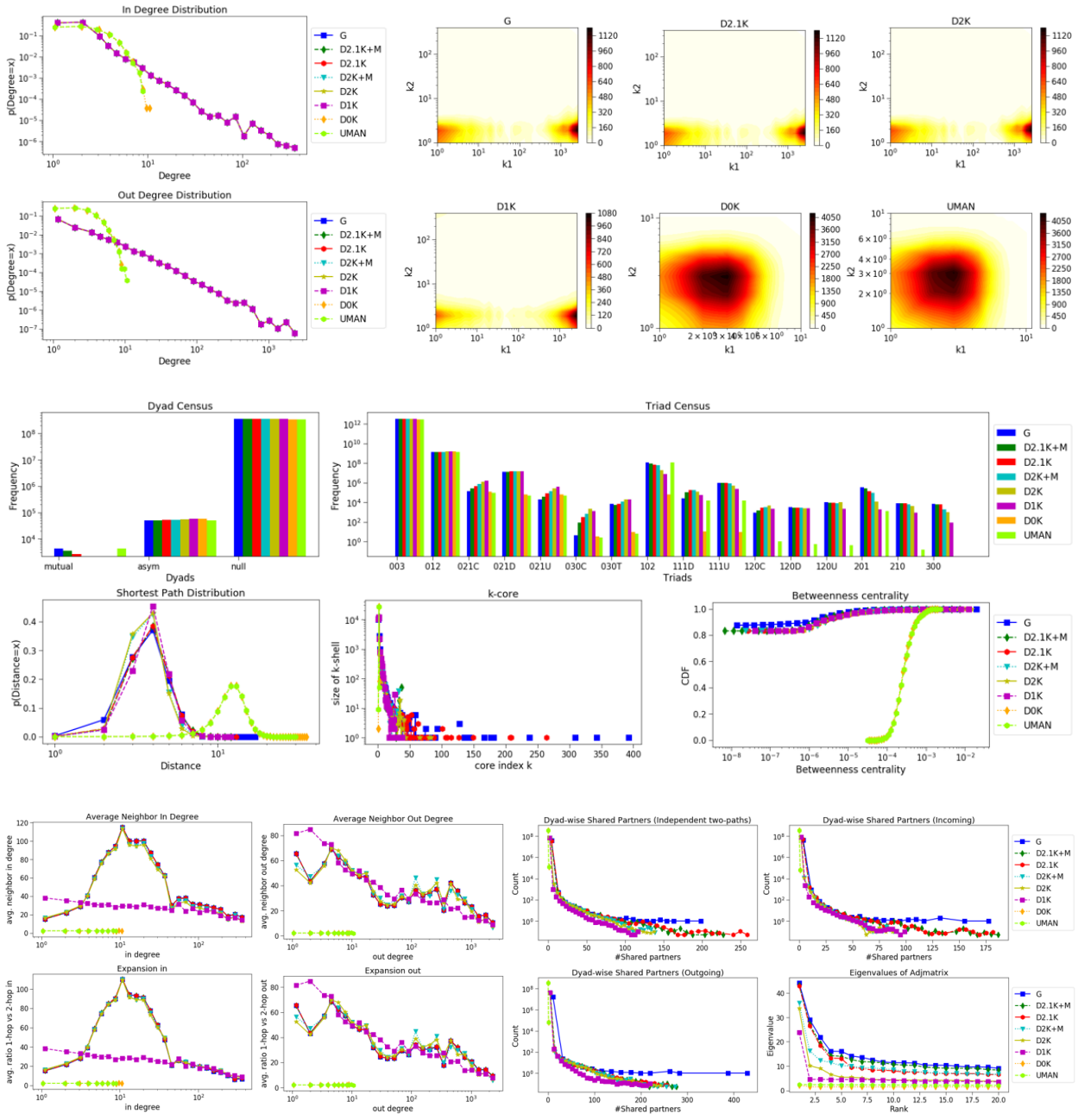


Figure 3.7: Results for AS-Caida graph: Directed Degree Distribution, Degree Correlation, Dyad-, Triad Census, Shortest Path Distribution, K-core distribution, Betweenness Centrality, Expansion, Average Neighbor Degree, DSP and top 20 Eigenvalues

of these networks have characteristics that makes them different from Twitter, *e.g.* p2p-Gnutella08 does not contain any mutual edges. The most interesting question is whether D2K or D2.1K capture network properties more accurately. The answer is yes in most cases, but it might not be a significant improvement in targeting certain properties.

Local structures are generally better captured by D2K and even more precisely for D2.1K, but global properties might not be significantly affected depending on the original network. However, this result is not surprising, since one of the main assumptions of the dK-series is that it is not necessary to target high d values for every graph [57].

3.5 DAG Construction

We build on and extend the dK-series [52] and directed dK-series [67]. Recall that 0K/D0K are ER graphs with fixed number of edges; 1K/D1K graphs have a specified degree sequence (DS or DDS, respectively); 2K/D2K graphs have a specified joint degree matrix. Also recall the inclusion property of dK-series: fixing 2K (JDM), fixes 1K (the degree sequence is the marginal of JDM), which in turns fixes 0K (the average node degree). Next, we provide analogous definitions, specifically for directed acyclic graphs (DAGs).

DAG0K: This is a trivial extension of the ER model for DAGs. We want to construct graphs with n nodes (in a target topological order) and m number of edges, s.t. edges only point from node v_i to v_j if $i < j$ in the topological order.

DAG1K: The input to this problem is an Ordered Directed Degree Sequence (ODDS), *i.e.,* a directed degree sequence (in and out-degree pairs) but the order of the sequence matters since this a DAG: every realization of an ODDS must be a DAG with a topological ordering that corresponds to the ODDS ordering. This definition is consistent with the notation from

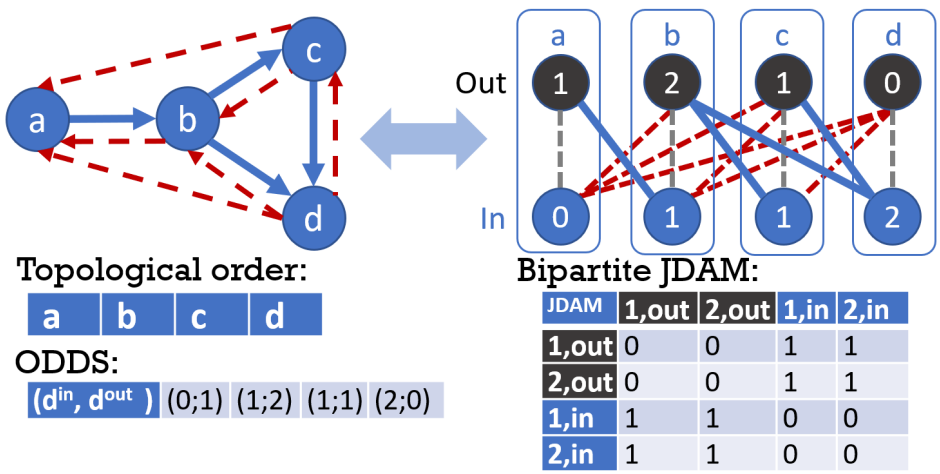


Figure 3.8: Specifying DAG1K and DAG2K: ODDS and bipartite representation with non-chords (dashed lines) defined by topological order.

Karrer and Newman [47].

$$ODDS = \{(d_1^{in}, d_1^{out}), (d_2^{in}, d_2^{out}), ..., (d_n^{in}, d_n^{out})\} \tag{3.1}$$

DAG2K. In this chapter, we go beyond ODDS and also target degree correlation of DAGs. We define the input of the DAG2K problem as a target ODDS *and* the target directed degree correlations. We represent directed degree correlations with a bipartite $JDAM$, where the partition corresponds to $(degree, in/out)$ pairs, as in [67] and as elaborated upon in the following example.

Fig. 3.8 shows an example DAG and its corresponding ODDS and bipartite representation of degree correlations. The topological order of this DAG, (a, b, c, d), defines ODDS: $\{(0;1), (1;2), (1;1), (2;0)\}$, *i.e.,* the ordered sequence of $(in; out)$ degrees of nodes in that order. The D2K of *any* directed graph (not only DAGs) can be described [67] as shown on the right side of Fig. 3.8: (i) by split every node v into v_i^{in}, v_i^{out} parts; (ii) no edge is allowed (non-chords) between the *in* and *out* parts of the same node i, in order to avoid self loops; (iii) edges are allowed only between *out* to *in* nodes, so that the bipartite graph can be mapped 1-1 to a directed graph; and (iv) the bipartite JDAM defines the number of

edges between $(degree, in/out)$ partitions. In order to represent specifically DAG2K in this framework, we also prohibit backward edges in the topological order: there can be no edges allowed between v_i^{out}, v_j^{in}, if $i > j$ in the ODDS order; these forbidden edges or "non-chords" are shown in dashed lines on the top right of the figure.

3.5.1 DAG1K Construction

The state-of-the-art in DAGs with a specified ordered directed degree sequence was the one proposed by Karrer and Newman [47] and reviewed in Section 1.3: a configuration model-based construction that allows multi-edges but no self-loops.

Here, we follow a different approach: using a network flow reduction for a directed degree sequence (DDS), originally suggested in [54]. We extend it to DAGs by incorporating non-chords corresponding to the topological order inherent in $ODDS$, as discussed in the previous section. Fig. 3.9 shows the flow network for the example of Fig. 3.8. More formally, for any target $ODDS$ with n nodes, we can define a flow network, F, that essentially extends the bipartite graph discussed in the previous section, as follows:

- Split every node $v_i, i = 1, ..., n$ that appears in $ODDS$ into v_i^{out}, v_i^{in}, with out and in degrees d_i^{out}, d_i^{in}, respectively.[7]
- Add directed edges from *out* to *in* nodes, as permitted by the ODDS order: *i.e.,* add edges $(v_i^{out}, v_j^{in}), i = 1, ..., n, j = i + 1, ..., n$. Set the capacity of all these edges to 1.
- Add source node, s, and connect it to all *out* nodes: add edges (s, v_i^{out}) with $capacity(s, v_i^{out}) = d_i^{out}, i = 1, ..., n$.
- Add sink node, t, and connect all *in* nodes to it: add edges (v_i^{in}, t) with $capacity(v_i^{in}, t) =$

[7]Nodes with in or out degree zero could be removed from the flow network, since they will not increase any flow. However, this won't improve the worst case running time.

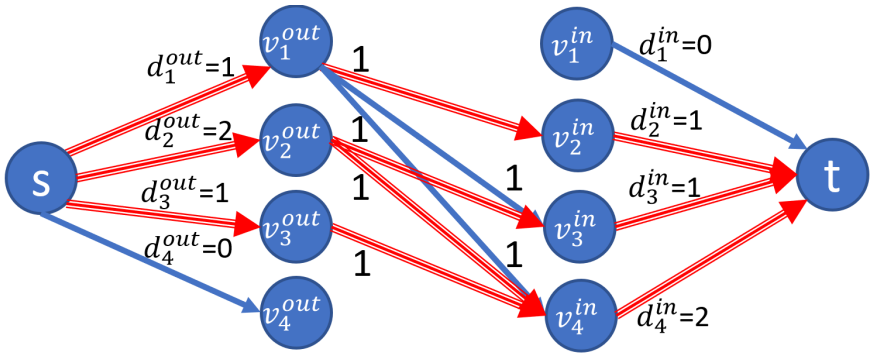


Figure 3.9: Example: flow network corresponding to ODDS from Fig. 3.8. Non-chords from $ODDS$ are the missing edges between v_i^{out} and v_j^{in}. The maximum flow is shown in red and it includes edges with $flow = capacity$.

$d_i^{in}, i = 1, ..., n.$

A trivial and necessary realizability condition for any ODDS is that the sum of in-degrees is equal to the sum of out-degrees.

Theorem 3.1. *Consider an ODDS with* $\sum_{i=1}^{n} d_i^{out} = \sum_{i=1}^{n} d_i^{in}$ *and construct the corresponding flow network* F. *The ODDS has a simple realization,* G, *if and only if* F *has a maximum flow value* $\sum_{i=1}^{n} d_i^{out}$.

Proof. $\rightarrow$ Assume that $ODDS$ has at least one simple realization, G, then we need to show that the maximum flow of F is equal to $\sum_{i=1}^{n} d_i^{out}$. Indeed, there exists a topological order of nodes $(\{a_1, ..., a_n\})$ in G that corresponds to the $ODDS$; then all edges in G, (a_i, a_j), have a corresponding pair, (v_i^{out}, v_j^{in}), in F. We set a flow in F using this topological order, as follows:

- $\forall i, flow(s, v_i^{out}) = d_i^{out}$, and $flow(v_i^{in}, t) = d_i^{in}$
- $\forall i, j : flow(v_i^{out}, v_j^{in}) = 1$, if edge (a_i, a_j) is present in G.

Since G is a realization of $ODDS$, there are exactly d_i^{out} outgoing edges set to $flow = 1$ for every v_i^{out} node in F and similarly there is exactly d_i^{in} incoming edges set to $flow = 1$ for

every v_i^{in} node in F. This means that the flow is valid. $\sum_{i=1}^{n} d_i^{out}$ is also the maximum flow value, since every edge out of s has $flow = capacity$.

$\leftarrow$ Now assume that the maximum flow of F has value $\sum_{i=1}^{n} d_i^{out}$ [8]; we need to show that $ODDS$ has at least one simple realization, G. In this case, every v_i^{out} node has exactly d_i^{out} outgoing edges set to $flow = 1$, similarly every v_i^{in} node has exactly d_i^{in} incoming edges set to $flow = 1$. We can construct a realization, G, first by removing s and t and then by merging nodes v_i^{out} to v_i^{in}, while only keeping edges between these nodes with $flow = 1$. By construction, F and G do not contain self-loops or edges violating the topological order. Since capacities between in and out nodes are 1, no multi-edges are necessary, hence G is a simple realization of $ODDS$. □

The **running time** depends on the network-flow algorithms and also the gadget's size. Our flow network uses $O(|V|)$ nodes and $O(|V|^2)$ edges. The running time is $O(|V|^3)$ using FIFO Push-Relabel max-flow algorithm [36], which can also be parallelized [7]. Network flow algorithms run in time that is comparable to degree sequence (DS, DDS) construction algorithms on dense graphs.

Tutte's gadget, reviewed in Section 1.3 can also be used to generate simple $ODDS$ realizations. However, its time complexity is order of magnitudes higher, since the gadget's size is $O(|V|^2)$ nodes and $O(|V|^3)$ edges. To construct a realization with $|V|$ nodes, we would need to solve a maximum cardinality matching problem on Tutte's gadget, that runs in $O(|V|^4)$ using the algorithm presented in [53]. The gadget's size gets larger as we decrease the density of the graphs. We have experimentally evaluated this trade-off on random DAG0K graphs and we show the results in Table 3.3.

Sampling from ODDS realizations is possible using MCMC methods, since the space of simple ODDS realizations are connected using ODDS-preserving double-edge swaps, as

[8] There is always an integral maximum flow assignment, since all capacities are integers.

Table 3.3: Running time of ODDS methods on random DAGs with different number of nodes and density. Reported average time is over 20 runs in seconds (s) and using a NetworkX -based implementation [40].

Graph Density	Network-flow 50 nodes	Tutte's gadget 50 nodes	Network-flow 100 nodes	Tutte's gadget 100 nodes
0.10	0.026s	9.989s	0.146s	178.020s
0.25	0.038s	8.284s	0.171s	153.932s
0.50	0.039s	5.776s	0.183s	114.155s
0.75	0.043s	4.107s	0.195s	81.899s
0.90	0.040s	2.969s	0.196s	51.041s

shown by Carstens [13]. This can be an alternative method to construct realizations by edge rewiring in cases when an input graph is available.

3.5.2 DAG2K Construction

We defined the input of DAG2K as a target ODDS *and* a target directed degree correlation together, and we formulated the problem as a **special case of D2K** [67], by imposing non-chords not only between the *in* and *out* parts of the same node, but also due to the topological order. However, DAG2K turns out to be significantly more challenging than D2K, and existing D2K algorithms from [67] do not solve general inputs of DAG2K, due to the multiple non-chords per node. Next, we review three other related approaches and explain why they cannot solve DAG2K in general.

First, we note that DAG2K is a **special case of the BPAM** (Bipartite Partition Adjacency Matrix) problem: non-chords are defined by $ODDS$ while the $JDAM$ input corresponds to the $BPAM$. However, BPAM has only been solved with high probability using a high order polynomial time algorithm [17], and there is no known deterministic and efficient algorithm yet.

A second approach is to run local search methods, such as **MCMC, starting from DAG1K**

realizations and targeting the desired DAG2K, for example by rewiring in the ODDS realizations or enumerating maximum flow assignments in the network-flow gadget. This approach is justified by the fact that (i) DAG2K graphs are a subset of DAG1K graphs and (ii) DAG1K graphs are known to be connected over double-edge swaps [13]. However, it would require the enumeration of all DAG1K graphs for DAG2K inputs that are not realizable.

Third, we considered a hypegraph extension to our network-flow gadget for DAG1K to directed hypergraphs for DAG2K. Fig. 3.10 shows an example based on the input of Fig. 3.8: we add new nodes to represent $JDAM$ and we replace edges with between v_i^{out}, v_j^{in} with hyper-edges to capture flow contributions to both degrees and $JDAM$. More formally, the **network flow hypergraph**, H, can be defined for given a $ODDS$ and $JDAM$ input as follows:

- Split every node $v_i, i = 1, ..., n$ that appears in $ODDS$ into v_i^{out}, v_i^{in}, with out and in degrees d_i^{out}, d_i^{in}, respectively.
- Add node $p_{k,l}$, for every pairs of part of the partitions where $\{k, out\}, \{l, in\}$.
- Add directed hyper-edges from out and $p_{k,l}$ to in nodes, as permitted by the ODDS order: add edges $(v_i^{out}, p_x, v_j^{in})$, $i = 1, ..., n, j = i + 1, ..., n, x = \{d_i^{out}, d_j^{in}\}$. Set capacities of these edges to 1.
- Add source node, s, and connect it to all out nodes: add edges (s, v_i^{out}) with $capacity(s, v_i^{out}) = d_i^{out}, i = 1, ..., n$.
- Connect s to all $p_{k,l}$ nodes: add edges $(s, p_{k,l})$ with $capacity(s, p_{k,l}) = JDAM(\{k, out\}, \{l, in\})$, for every $\{k, out\}$, $\{l, in\}$ pairs.
- Add sink node, t, and connect all in nodes to it: add edges (v_i^{in}, t) with $capacity(v_i^{in}, t) = d_i^{in}, i = 1, ..., n$.

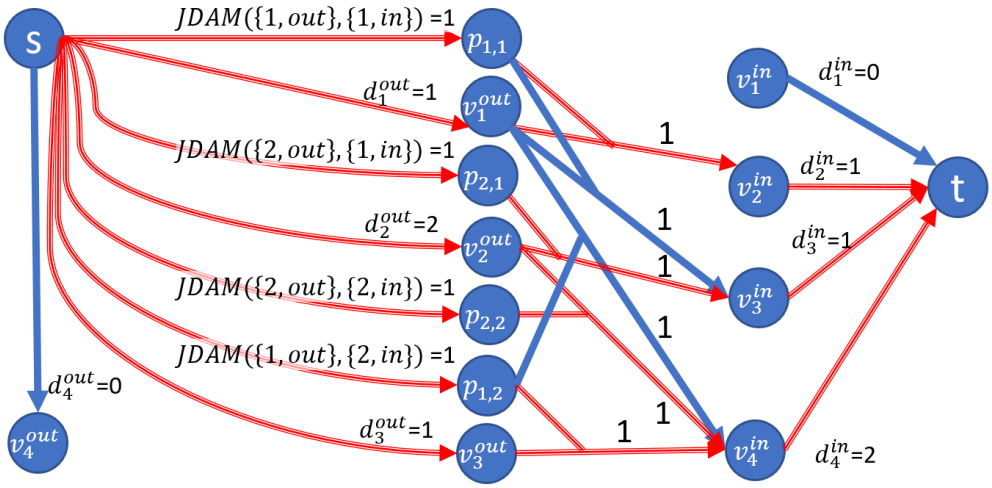


Figure 3.10: Network Flow HyperGraph for DAG2K using input from Fig. 3.8. The maximum flow is shown in red ($flow = capacity$).

When H has an integral maximum-flow solution, it is easy to see using the proof technique in previous section, whether DAG2K is realizable or not as a simple DAG. However, max flows in hypergraphs could have non-integral values.[9] We tried rounding non-integral solutions and found that (i) the target degree correlation or the ODDS can be easily achieved but (ii) it remains open to find a rounding scheme that achieves both.

Although none of the aforementioned approaches can solve DAG2K in general, they can still be used to solve special DAG2K inputs (*e.g.,* DAG2K hypergraphs with integral max flow value) or as heuristics without guarantees of deterministic construction or running time (*e.g.,* MCMC, BPAM). Next, we turn our attention to a subset of realizable D2K inputs, whose every realization is a DAG and existing D2K algorithms can be used for construction.

[9] H is a Leontief Directed Hypergraph (LDH) as defined in [14], more precisely a 2-LDH, since every edge has at most two tails and one head. A 2-LDH has a totally unimodular incident matrix (and an integral extreme point solution to the corresponding linear programming) if and only if it does not have odd pseudocycles. However, H could have odd pseudocycles as defined in [14], and we can also confirm through simulations that it is easy to generate incident matrices for H that are not totally unimodular.

3.5.3 D2K+L: Level Graphs

We now consider a special interesting case of DAGs: directed acyclic graphs with level assignment: every node is assigned a level and edges only point from earlier to later levels.[10] Level graphs can be used to create a "compact" visualization of DAGs and have been studied in the graph drawing community [10], [6]. Level assignments can be modeled by adding non-chords, not just between v_i^{out} and v_j^{in} if $i > j$ in $ODDS$, but also between nodes in the same level. This can be incorporated into $DAG1K$ and our network-flow reduction still holds.

Defining D2K+L. Next, we show that D2K+L is a special case of D2K (whose input is DDS and JDAM) with additional attributes to further partition nodes into levels. The node partition in the bipartite representation corresponds to the node's degree, in/out label and node's level assignment. Fig. 3.11 shows an example input of D2K+L. We represent degrees with letters k, l, level assignments with i, j, and parts of the partition as $\{k, in, i\}$ or $\{l, out, j\}$.

Realizability: The sufficient conditions for such D2K+L input to be realizable as DAGs are the following:

- Realizability conditions hold for D2K input [67].
- *and* the $JDAM$'s reduced skeleton graph, RSG, is a DAG.

Construction: Since D2K+L is a special case of D2K (with further level partitioning), the D2K algorithm presented in [67] can be used to construct a simple realization. We observe that if RSG is a DAG, then non-chords only appear where $JDAM(\{k, in, i\}, \{l, out, i\}) = 0$ and every realization is a DAG; see Lemma 3.2. This allows us to use JDAM algorithms for

[10] Please note that level assignment is more restrictive than topological ordering, since it prohibits edges within levels with multiple nodes. Topological ordering is a level assignment with $k = n$ levels, *i.e.* single node per level.

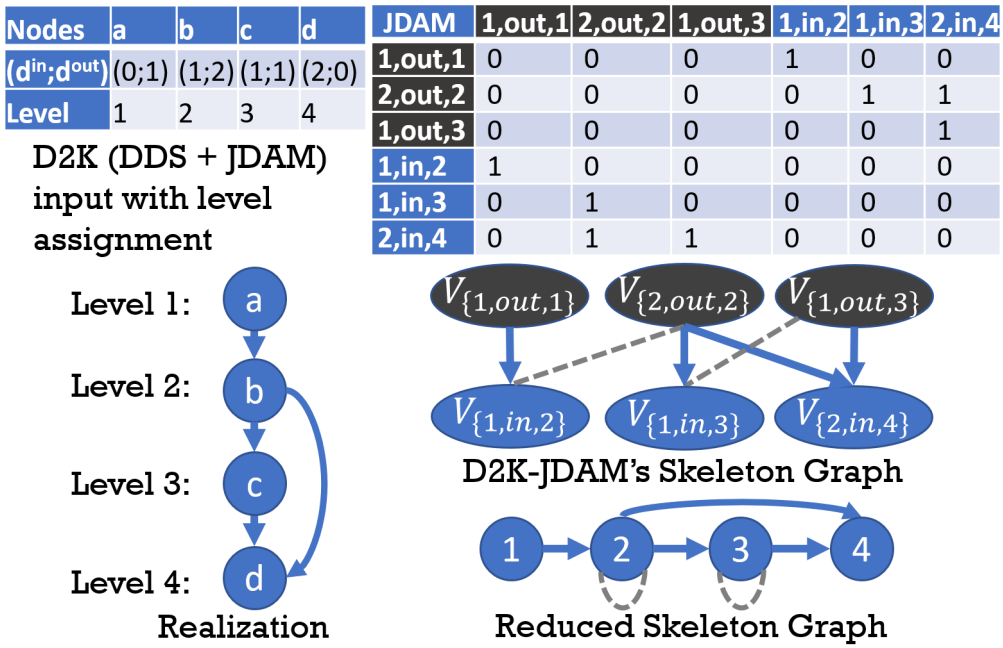


Figure 3.11: D2K+L input example: ODDS and levels, the corresponding JDAM, Skeleton Graph, Reduced Skeleton Graph and realization.

construction and sampling using double-edge swap-based MCMC or importance sampling algorithm.

To prove the above statements, we need to extend the definition of Skeleton Graphs from [24] to the D2K case, and we define the Reduced Skeleton Graph (RSG) of a D2K's JDAM. Fig. 3.11 shows an example of transforming a specific input JDAM to *RSG*.

Definition 3.1. *We define the Skeleton Graph of D2K's JDAM, where each part of a partition* $\{k, in/out, i\}$ *for every degree* k *and level assignment* i *has a node and a directed edge from* $\{l, out, j\}$ *to* $\{k, in, i\}$ *if* $JDAM(\{l, out, j\}, \{k, in, i\}) > 0$.

Definition 3.2. *We define the Reduced Skeleton Graph (RSG) of a JDAM as the directed graph representation of D2K's JDAM, where nodes in the Skeleton Graph of D2K's JDAM with the same level assignment* i *(e.g,* $\{k, in, i\}$, $\{l, out, i\}$*) are merged into node* i*; we preserve the direction of edges but remove multi-edges.*

Lemma 3.2. *If RSG is a DAG, then every realization of the D2K input is also a DAG.*

Table 3.4: Test citation networks from SNAP [49] with their sizes and D2K+L average construction time (over 20 runs).

Dataset Name	#Nodes	#Edges	Generation time
`Cit-HepTh`	27,751	351,500	8.5 sec
`Cit-HepPh`	34,529	419,528	10.6 sec
`cit-Patents`	2,745,762	13,965,410	12.22 min

Proof. Let us assume that there is at least one realization, G, that is not a DAG, *i.e.,* it has a cycle C. Edges along this cycle, C, must contribute to entries to the corresponding $JDAM$ and its skeleton graph, SG. When we merge SG nodes during the RSG construction, we merge nodes from G with the same attribute values. C will be preserved as a cycle in RSG iff nodes along C had different attribute values. RSG might have multiple cycles (and self-loops) when C has fewer attribute values than nodes. In both cases, this leads to a contradiction since we assumed RSG is a DAG. □

3.6 Simulations for Real-World Directed Acyclic Graphs

Setup. We experimentally evaluate our DAGdK algorithms when targeting real-world citation networks available from SNAP [49]. We had to do pre-processing to ensure that our test graphs were indeed DAGs: (i) we removed edges that appeared to create cycles (based on publication date) from the `Cit-HepTh` and `Cit-HepPh` networks; (ii) we removed nodes that did not have date from the `cit-Patents` network. Next, we report the final graph sizes and the construction time for D2K+L (averaged over 20 realizations).

For each of the three networks, we first compute $ODDS$ and we assign a level for each node using the Longest-Path algorithm [10] to compute D2K+L. Then, we generate 20 different realizations, using DAG0K, DAG1K and D2K+L inputs for each graph. We implemented D2K+L based on the JDM implementation from NetworkX [40]. We found that our network-flows did not scale beyond 10,000 nodes for DAG1K using the NetworkX implementation,

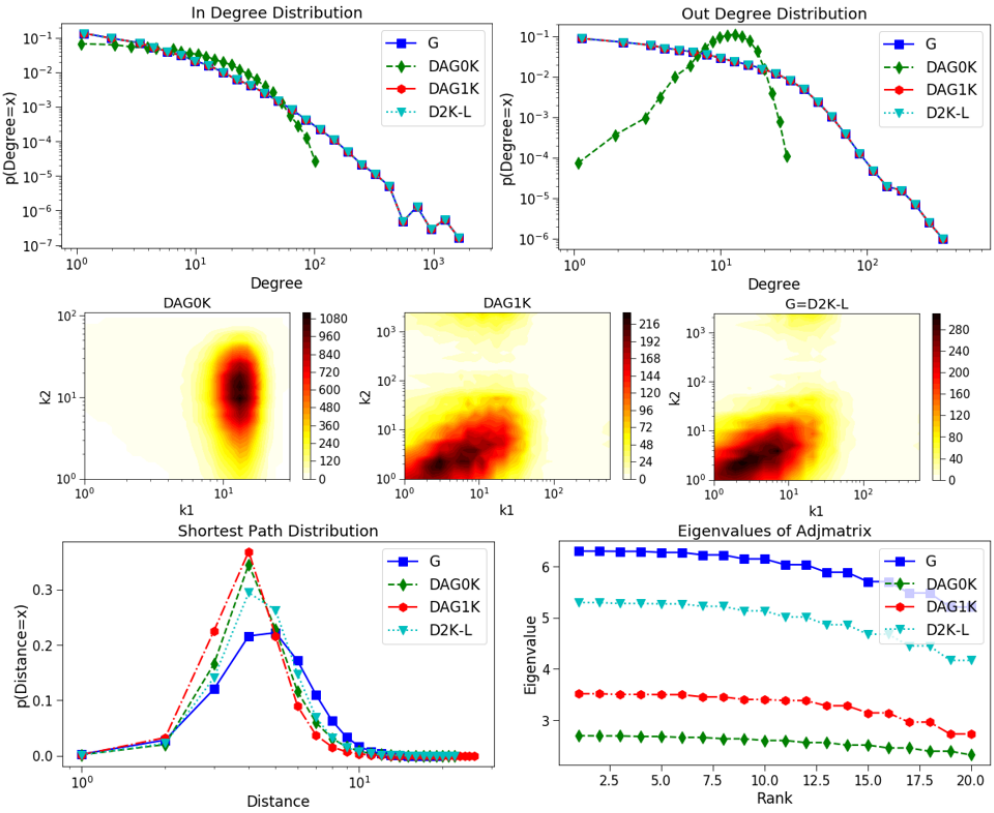


Figure 3.12: `Cit-HepTh` results

thus could not finish the construction of our test graphs. We have used MCMC sampling for ODDS with $O(|E|)$ steps starting from the input graph to generate DAG1K instances for evaluation.

Effect of Level Assignment on the resulting DAG. The Longest-Path algorithm assigns the minimum number of levels [10], but it does not minimize the size of the overall partition defined by the $JDAM$. This is due to the fact that $JDAM$ is based on the combination of level assignment and unique degrees as discussed in Section 3.5.2. In the next table, we report the number of levels, unique in/out degree groups and size of the partition generated. For the first two smaller graphs, parts of the partition are smaller (average size: 2.92-4.66), making it more likely to fix subgraphs. `Cit-Patents` has only 31 levels assigned resulting in a partition into larger parts (average size: 873.09), which leaves more flexibility to construct realizations from a larger space.

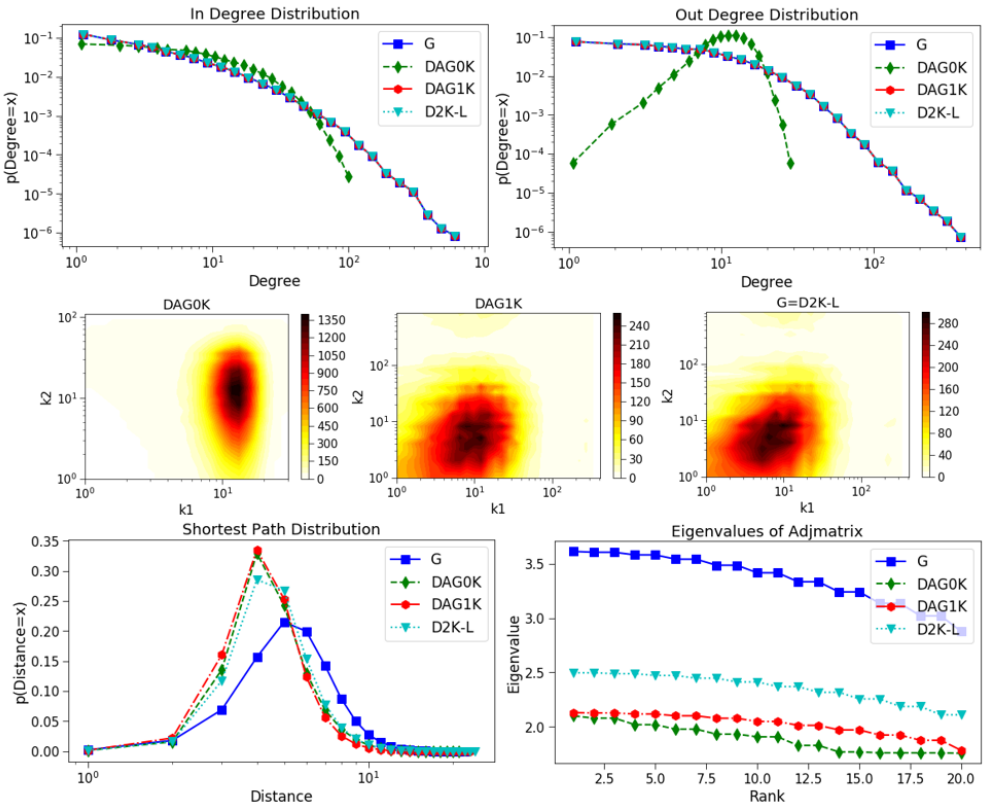


Figure 3.13: `Cit-HepPh` results

Table 3.5: Effects of level assignments: number of levels assigned, number of unique in/out degrees, resulting partition size and average part size.

Name	#Levels	#In-deg.	#Out-deg.	Partition size	Avg. Part size
`Cit-HepTh`	302	282	153	16454	2.92
`Cit-HepPh`	167	264	166	12919	4.66
`cit-Patents`	31	256	324	4905	873.09

Network Properties of generated DAGs. In Fig. 3.12,3.13,3.14, we report results for the three citation networks. We report several graph properties, some of them explicitly targeted during construction (degree distribution, degree correlation) and some of them not (shortest path distribution and leading eigenvalues); we compared these properties between the synthetic and the original graphs. The degree distributions and degree correlations were explicitly targeted and exactly achieved by DAG1K and D2K+L, respectively, as expected. The DAG1K graphs are better than DAG0K graphs, but do not achieve the target degree correlation; the relatively small difference from target can be due to how long the MCMC

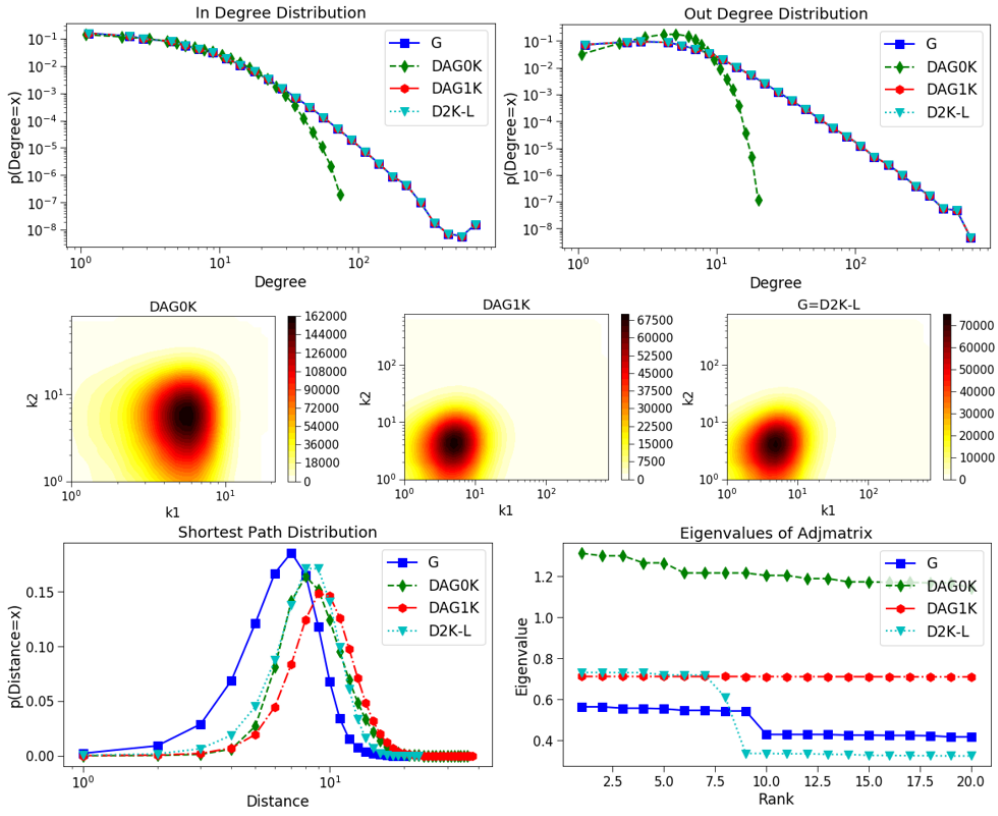


Figure 3.14: `cit-Patents` results

sampler was running in our experiments. W.r.t. to other properties that were not explicitly targeted by our DAGdK algorithms (shortest paths, leading eigenvalues, and others not reported due to lack of space, such as triad census, average neighbor degree, betweenness centrality), we saw that they are better approximated for higher d. In summary, we confirm that the DAGdK-series converges to the (targeted and non-targeted) properties as d increases.

3.7 Summary

We have extended the undirected dK-series to directed graphs by defining directed 2K problem (D2K): the input of directed degree sequence and degree correlations together. We have shown the necessary and sufficient conditions to decide realizability and we have developed

an efficient algorithm based on techniques from Chapter 2. We have also shown that the importance sampling algorithm defined by Bassler *et al.* [9] can be applied to D2K as well. Finally we have shown several extensions of D2K (such as D2.1K, number of mutual edges and balanced realizations) and we evaluated the convergences of the directed dK-series on real-world graphs.

In addition, we presented methods for generating DAGs with a prescribed ordered degree sequence and degree correlations. For the DAG1K problem, we provided a new algorithm that is orders of magnitude faster than previous approaches. For DAG2K, we showed that previous approaches do not apply but we also identified a family of D2K inputs that admit only acyclic realizations, thus previously known D2K algorithms can efficiently construct and sample these DAGs. We evaluated the effectiveness of our algorithms in constructing synthetic DAGs that resemble real-world citation networks. Along the way, we made connections between several graph construction problems, some of which were previously disconnected, which can hopefully enable the (re)use of algorithms.

Chapter 4

Graph Construction from Embeddings

4.1 Introduction

Independently of the previously discussed graph construction literature, there is also a growing body of work on *graph embeddings*, or latent representations of graphs, which are widely used for tasks such as graph reconstruction, link prediction, and node classification. For example, in link prediction, we may be able to observe some of the edges of the graph, and we want to predict the missing ones; or we may want to predict new edges that are likely to be formed over time. In node classification, we may know the attributes of some nodes, and want to classify the remaining ones based on the graph structure. In graph reconstruction, the goal is to create a graph with maximum edge overlap with the original graph. Fig. 4.1 depicts an overview of the tasks.

In this chapter, we propose a new approach that bridges the gap between these two previously disconnected bodies of work: the classic graph construction models and more recent notions of graph embeddings from machine learning. This leads to a more general model than either of the two approaches alone, and allows us to leverage and combine existing techniques and

applications for a variety of tasks.

More specifically, we are interested in representations that can be used for generating graphs that exhibit target local properties. To that end, we define the Neighborhood Partition Matrix (NPM), which captures the local neighborhood structure of a graph in the following sense. Given an *arbitrary node partition* $\mathcal{P}$, $NPM[i, j]$ specifies the number of edges from node i to nodes in part j of $\mathcal{P}$. We then define the NPM construction problem as follows: given an input NPM, generate simple graph realization(s) that have that NPM exactly, if such realizations exist. Fig. 4.1 presents an overview of our intended use of NPM: given a real graph with a defined node partition $\mathcal{P}$, the NPM of the real graph is measured. We use this given NPM as input to perform different tasks, namely: graph construction, graph reconstruction, link prediction, and node classification.

We note that NPM generalizes node partitioning by degree only (which was the case *e.g.* in the Degree Spectra Matrix (DSM) [9] and in the Neighborhood Degree List (NDL) [8]) to arbitrary node partitions. This generalization allows us to pose, for the first time, NPM construction as a graph embedding problem with arbitrary dimensions, where each dimension corresponds to a part in the partition. This results in an interpretable embedding based on the meaning of the node partition. We demonstrate the generality of the NPM framework, by considering different strategies of partitioning nodes to compute NPM. We also establish connections between NPM and other graph construction problems, which we leverage to decompose the NPM problem into a union of degree sequence problems. This, in turn, enables us to solve NPM efficiently for realizability and construction, and to provide approaches for sampling. Furthermore, we discuss extensions of NPM to capture other properties: (i) for clustering coefficients, we prove the NP-Hardness of NPM with prescribed number of triangles per node; (ii) for NPM with directed graph construction and NPM with non-chords (*i.e.* forbidden edges), we show how to solve them efficiently.

Compared to other graph embeddings, NPM has two qualitative advantages: (i) it creates

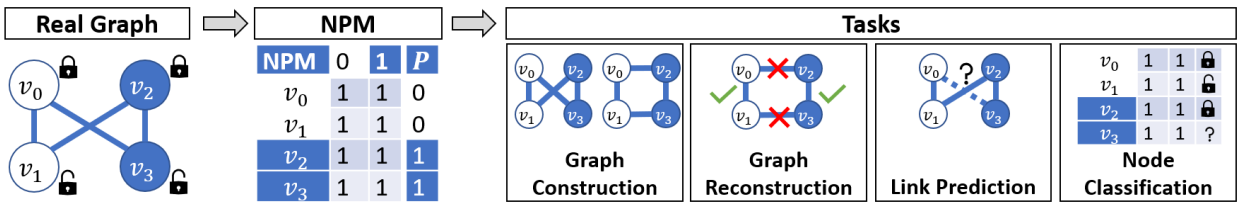


Figure 4.1: An overview of our approach: given a real graph, the NPM is measured for a partition $\mathcal{P}$, to capture the graph's local neighborhood structure. This NPM is the input to our construction problem and it is used in the following tasks: (1) Graph Construction, (2) Graph Reconstruction, (3) Link Prediction, (4) Node Classification. (The icon of Lock is used to represent an example node attribute to be used as a binary label for the node classification task.)

graph realizations that exhibit exactly the target properties (such as degree sequences, degree correlations, etc); (ii) it is more interpretable. NPM also has some technical differences, *i.e.* the embedding vectors have integer instead of continuous values. Compared to graph construction, the decomposition of NPM leverages efficient approaches but generalizes beyond partitioning by degree only. In our evaluation, we show that NPM outperforms baseline graph embedding methods for graph construction and graph reconstruction, for several datasets, and performs comparable to baselines on link prediction and node classification tasks. Thus, the NPM model brings qualitative improvements (flexibility and interpretability), while also achieving better or – in the worst case – similar performance w.r.t. all tasks.

The outline of the rest of the chapter is as follows. Section 4.2 provides additional background for this chapter. Section 4.3.1 defines the Neighborhood Partition Matrix construction problem and introduces the main concepts. Section 4.3.2 extends NPM construction to include additional constraints, such as local clustering, directed graphs and non-chords. Section 4.3.3 describes the tasks (graph construction, graph reconstruction, link prediction, node classification) and the respective metrics for evaluation. Section 4.4 evaluates the performance of NPM with different partitions against baseline methods. Section 4.5 concludes the chapter.

4.2 Background on Graph Embeddings

Graph embeddings are latent variable models that capture low dimensional representations of graphs. They are used to perform machine learning tasks such as node classification, link prediction, clustering or visualization of embeddings to highlight underlying structure of graphs; some of these tasks are depicted on Fig.4.1. Using the definition from [38]:

Definition 4.1. *Given a graph $G(V, E)$, a graph embedding is a mapping $f : v \to y \in R^d$ $\forall v \in V$, s.t. $d \ll |V|$ and the function f preserves some proximity measure defined on G.*

In the graph embedding framework, graphs are usually generated by using an operator (such as dot product) of two embedding vectors between all pairs of nodes and by returning top recommended edges. In contrast to the graph construction methods, the constructed graphs might only have certain properties in expectation. The interpretation of graph construction inputs is clear from their targeted properties, while embedding dimensions usually do not carry specific meaning. However, graph embeddings are easier to extend to weighted graphs, which is a missing property of most studied graph construction models. Last but not least, graph embeddings are the underlying model for important machine learning tasks such as node classification, link prediction, etc.

Graph embeddings are rooted in the well-studied matrix representations such as *Locally Linear Embeddings* [61] or *Laplacian Eigenmaps* [11], that use the eigen structure of the adjacency matrix (or a derived matrix) and take eigen vectors as the embedding or other matrix factorization methods [2]. *HOPE* [58] is a method that attempts to preserve higher order proximities of nodes in terms of similarity matrices and uses generalized Singular Value Decomposition to scale more efficiently. More recently, random walk-based methods become preferred such as *DeepWalk* [59] or *node2vec* [39], where embeddings are learned based on sequences generated by random walks in local neighborhoods of nodes.

4.3 Proposed Framework: NPM

4.3.1 The Neighborhood Partition Matrix (NPM) Problem

In this section, we present a new approach for generating graphs with a target local neighborhood structure. To capture that structure, we define the Neighborhood Partition Matrix (NPM), of a graph G, as follows:

Definition 4.2. *Given an undirected graph $G(V, E)$ and an arbitrary partition $\mathcal{P}$ of the nodes into d parts, NPM of G is a $|V| \times d$ matrix, such that $NPM[i, j]$ specifies the total number of edges from node i to nodes in part j of $\mathcal{P}$.*

A strength of our proposed NPM model is the flexibility it provides to define arbitrary partitions. We considered the following list of partitions: random partitions into d parts; degrees; community detection algorithms from literature such as [56]; k-core decomposition; connected components. The Degree Spectra Matrix (DSM) [9] and Neighborhood Degree List (NDL) [8] are special cases of NPM with a node partition by degrees. Furthermore, arbitrary partitions allow us to interpret NPM as a graph embedding. This bridges the gap between two previously disconnected bodies of literature: the classical graphical construction and graph embeddings, while leveraging existing techniques and applications of both.

We define the NPM graph construction problems as follows:

- **Realizability:** Given a target NPM and $\mathcal{P}$, decide whether it is realizable, *i.e.* whether there exist a *simple* graph with this exact NPM, when nodes are partitioned by $\mathcal{P}$.
- **Construction:** Design an algorithm that constructs at least one graph realization with the target NPM and $\mathcal{P}$.
- **Sampling:** Sample from the space of all graph realizations that exhibit the target

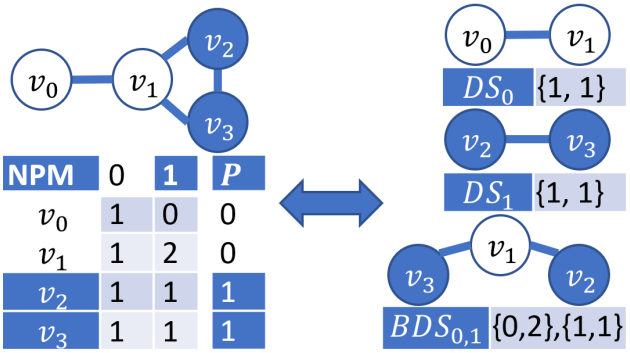


Figure 4.2: An example for the decomposition of NPM into degree sequence problems (DS_i, $BDS_{i,j}$) for a simple graph with its nodes partitioned into two parts ($V_0 = \{v_0, v_1\}, V_1 = \{v_2, v_3\}$). The union of the edges from the degree sequence realizations return the input graph.

NPM and $\mathcal{P}$.

Realizability and Construction

We show the decomposition of the NPM problem into a union of independent degree sequence problems as shown in Algorithm 4.1 and Fig. 4.2:

- DS_i is the degree sequence problem for nodes in part i (V_i), *i.e.* the degrees of nodes in V_i towards V_i.
- $BDS_{i,j}$ is the bipartite degree sequence problem for nodes in parts i, j (V_i, V_j), *i.e.* the degrees of nodes in V_i towards V_j and the degrees of nodes in V_j towards V_i.

Theorem 4.1. *An NPM input is realizable if and only if every degree sequence problem (DS_i and $BDS_{i,j}$) is realizable.*

Proof. If NPM is realizable, then any realization of an NPM also provides a realization for the degree sequence problems by taking the induced subgraph by V_i for DS_i or by V_i, V_j for $BDS_{i,j}$ without the edges within the same parts (excluding edges between V_i nodes or V_j nodes).

Algorithm 4.1 NPM Construction

Given NPM, $\mathcal{P}$
Create graph $G(V, E)$ and nodes: $v \in V_i$ for each part i in $\mathcal{P}$
for every pair of i, j parts of $\mathcal{P}$:
  if $i = j$:
    # Unipartite degree sequence problem (DS)
    $DS_i = \{NPM[v, i] | v \in V_i\}$
    $G_i(V_i, E_i) \leftarrow$ realization of DS_i
    $E \leftarrow E \cup E_i$
  if $i \neq j$:
    # Bipartite degree sequence problem (BDS)
    $BDS_{i,j} = \{NPM[v, j] | v \in V_i\}, \{NPM[v, i] | v \in V_j\}$
    $G_{i,j}(V_i \cup V_j, E_{i,j}) \leftarrow$ realization of $BDS_{i,j}$
    $E \leftarrow E \cup E_{i,j}$
return G

Conversely, if the degree sequence problems are realizable, then the union of any realization of the degree sequence problems returns an NPM realization. This is trivial to see, since we construct these degree sequences from the NPM with the correct degree for each node to every partition. □

The decomposition of NPM is similar to DSM [9] and NDL [8], but we make the crucial observation that Theorem 4.1 is independent of the partition of nodes. Algorithm 4.1 is parallelizable, since NPM has $O(\binom{d}{2})$ independent degree sequence problems, where realizability is solved by applying Erdős-Gallai [23] and Gale-Ryser theorems [30] and construction is done using Havel-Hakimi algorithms [42, 41, 30].

Sampling realizations

Importance sampling algorithms exist for the unipartite and bipartite degree sequence problems from NPM which run in $O(|V||E|)$ [12, 18, 48]. In addition, it is also possible to sample NPM realizations using MCMC approaches. For this purpose, we need to show that the space of NPM realizations is connected over a simple edge-rewiring. We define NPM-preserving

double-edge swaps as changing edges $\{(u,v),(w,x)\}$ to $\{(u,x),(w,v)\}$ where (1) u,v,w,x are four distinct nodes (to avoid self-loops) and (2) $(u,x),(w,v)$ were not edges before (to avoid multi-edges) and (3) u,w are in the same part of the partition and v,x are from the same partition, possibly different from u and w's part. It is clear that such a swap will preserve the degree of each node to the appropriate parts of the partition.

Theorem 4.2. *The space of NPM realizations are connected over NPM-preserving double-edge swaps.*

Proof. We use the decomposition of NPM into degree sequences from Theorem 4.1: given two realizations of NPM and their respective induced subgraphs for the degree sequence problems, these subgraphs are connected over degree-preserving double-edge swaps [64], [25]. However, the degree-preserving double-edge swaps are also NPM-preserving double-edge swaps, hence the union of these swaps will connect any two NPM realizations. □

Relation to Embeddings

Regardless of the partition defined, there is a clear interpretation of the resulting NPM. A node's integer vector from the NPM describes how many edges connect to different parts of the partition. This is different from other embedding methods where it is usually a real embedding vector that points to a position in a d-dimensional space. Structural similarity of nodes is present in our method, since two nodes with exactly the same vectors will be connected to the same partitions the same number of times.

NPM encodes node level constraints, but graph construction algorithms have also implicit constraints for edges during construction, when we restrict NPM realizations to be simple graphs. These implicit constraints make NPM different from most graph embedding methods, where pairwise operators of embedding vectors are independent of each other. This leads to a limitation of NPM as a graph embedding, namely that an entire graph has to

be constructed multiple times to compute an edge probability. While NPM construction is scalable and highly parallelizable, it is admittedly more complex than running a simple operator on two embedding vectors.

Finally, in the context of graph embeddings, it is desirable to partition the nodes into exactly d parts. If the selected partitioning method for NPM is not flexible to produce partitions with exactly d parts, we use a partition agnostic heuristic to change the partition to have d parts, as shown in Algorithm 4.2.

Algorithm 4.2 Heuristic for Partitions with d parts

Given $\mathcal{P}$, d
while Number of parts of $\mathcal{P} \neq d$:
  if Number of parts of $\mathcal{P} > d$:
    Merge two smallest parts of $\mathcal{P}$

  if Number of parts of $\mathcal{P} < d$:
    Split largest part of $\mathcal{P}$ into two parts

return $\mathcal{P}$

4.3.2 Extensions of NPM

We extend the basic NPM model to target additional structural properties such as local clustering coefficients, directed NPM and NPM with non-chords (*i.e.* edges not allowed to be present in any realization).

NPM with local clustering coefficients

In the context of dK-series, it has been shown that realizability of JDM with target number of triangles is an NP-Complete problem [19]. Here, we show that the realizability of NPM with local clustering coefficients, *i.e.* using an additional dimension to capture number of triangles attached to each node, is also NP-Complete. We adapt the proof strategy showed in

[19] for JDM with target number of triangles, and we use the well-known Graph 3-Coloring problem [31] in our reduction.

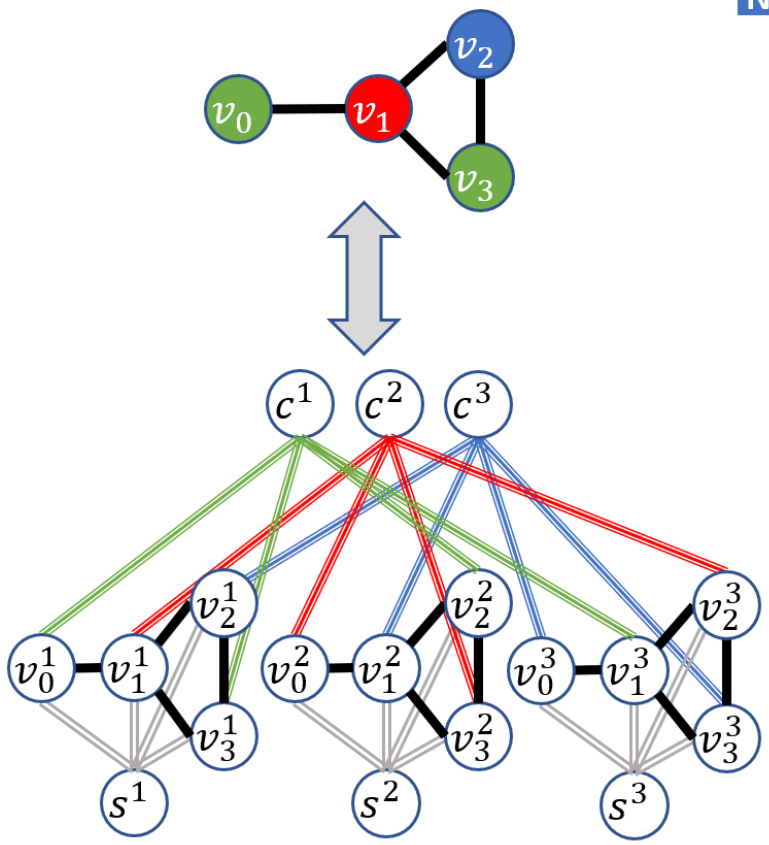
The input to this version of the NPM is the previously defined NPM with partition $\mathcal{P}$ and the number of triangles, t_i, attached to node i. The output for the decision version is whether there exist a simple realization with these constraints.

Theorem 4.3. *Realizability of NPM with prescribed number of triangles per node is NP-Complete.*

Proof. This problem is NP-Hard by Lemma 4.1 and Lemma 4.2, and a realization serves as a verifiable witness by measuring the NPM instance in polynomial time. □

We define the following NPM for a given graph, $G(V, E)$, as shown in Fig. 4.3:

- For each node $v \in V$, use three nodes: v^1, v^2, v^3; use coloring nodes: c^1, c^2, c^3 and stabilizing nodes: s^1, s^2, s^3.
- Create a partition of nodes as $\mathcal{P}_v = \{v^1, v^2, v^3\}$ for each node $v \in V$, and $\mathcal{P}_c = \{c^1, c^2, c^3\}$, $\mathcal{P}_s = \{s^1, s^2, s^3\}$.
- For $i = 1, 2, 3$, if $(u, v) \in E$, set $NPM[u^i, \mathcal{P}_v] = 1$ and $NPM[v^i, \mathcal{P}_u] = 1$, and for each node $v \in V$: $NPM[v^i, \mathcal{P}_c] = 1$, $NPM[v^i, \mathcal{P}_s] = 1$, $NPM[c^i, \mathcal{P}_v] = 1$, $NPM[s^i, \mathcal{P}_v] = 1$, and remaining NPM entries are 0.
- For each node $v \in V$, $t_{v^i} = t_v + deg(v)$ for $i = 1, 2, 3$; t_v is the number of triangles attached to node v in G.
- For each coloring node c, $t_{c^i} = 0$ for $i = 1, 2, 3$.
- For each stabilizing node s, $t_{s^i} = |E|$ for $i = 1, 2, 3$.



NPM	0	1	2	3	c	s	P	T
v_0^1	0	1	0	0	1	1	0	1
v_0^2	0	1	0	0	1	1	0	1
v_0^3	0	1	0	0	1	1	0	1
v_1^1	1	0	1	1	1	1	1	4
v_1^2	1	0	1	1	1	1	1	4
v_1^3	1	0	1	1	1	1	1	4
v_2^1	0	1	0	1	1	1	2	3
v_2^2	0	1	0	1	1	1	2	3
v_2^3	0	1	0	1	1	1	2	3
v_3^1	0	1	1	0	1	1	3	3
v_3^2	0	1	1	0	1	1	3	3
v_3^3	0	1	1	0	1	1	3	3
c^1	1	1	1	1	0	0	c	0
c^2	1	1	1	1	0	0	c	0
c^3	1	1	1	1	0	0	c	0
s^1	1	1	1	1	0	0	s	4
s^2	1	1	1	1	0	0	s	4
s^3	1	1	1	1	0	0	s	4

Figure 4.3: Example graph with 3-coloring and a realizable NPM input with prescribed number of triangles per node.

Lemma 4.1. *If G is 3-colorable (red, green, blue), then NPM with prescribed number of triangles per node is realizable.*

Proof. Given G, we perform the following steps:

- Copy G three times: $G^1(V^1, E^1)$, $G^2(V^2, E^2)$, $G^3(V^3, E^3)$ and add three coloring (c^1, c^2, c^3) and stabilizing nodes (s^1, s^2, s^3).
- Define partition $\mathcal{P}$ where parts correspond to the copies of nodes of G: $\mathcal{P}_v = \{v^1, v^2, v^3\}$ for each node $v \in V$, and $\mathcal{P}_c = \{c^1, c^2, c^3\}$, $\mathcal{P}_s = \{s^1, s^2, s^3\}$.
- Add edges (s^i, v^i) for all $v \in V$ and $i = 1, 2, 3$.

- For a red node in G, v_r, add edges to connect v_r's copies to coloring nodes: $(v_r^1, c^1), (v_r^2, c^2), (v_r^3, c^3)$.
- For a green node in G, v_g, add edges to connect v_g's copies to coloring nodes: $(v_g^1, c^2), (v_g^2, c^3), (v_g^3, c^1)$.
- For a blue node in G, v_b, add edges to connect v_b's copies to coloring nodes: $(v_b^1, c^3), (v_b^2, c^1), (v_b^3, c^2)$.

We show that the union of G^1, G^2, G^3 with the coloring and stabilizing nodes is a realization of the NPM defined earlier. It is clear that the partition and the number of nodes is correct. We have to confirm that all the edges are captured by the NPM and the embedding vectors are correct: since every edge $(u, v) \in E$ is copied three times, $NPM[u^i, \mathcal{P}_v] = 1$ (and $NPM[v^i, \mathcal{P}_u] = 1$ for $i = 1, 2, 3$). In our construction, every coloring node connects to exactly one copy of a node from G ($NPM[c^i, \mathcal{P}_v] = 1$ for $i = 1, 2, 3$). Similarly, every stabilizing node connects to exactly one copy of a node from G ($NPM[s^i, \mathcal{P}_v] = 1$ for $i = 1, 2, 3$). We have not added any other edges, hence the remaining entries of NPM are 0.

We make three observations to show that the constructed graph has the target number of triangles per node: (1) a valid coloring implies that coloring nodes do not form triangles, (2) a stabilizing node connects to every node in the same copy and creates $|E|$ triangles, (3) nodes in a copy of G preserve the attached triangles and get $deg(v)$ additional triangles from the stabilizing nodes. This takes into account all edges and triangles in the graph and exactly matches the input. □

Lemma 4.2. *If NPM with prescribed number of triangles per node is realizable, then G is 3-colorable.*

Proof. We denote a realization of NPM as G^*. We argue that G^* contains 3 copies (G^1, G^2, G^3) of G as subgraphs. Each stabilizing node connects to every part (except col-

oring) with one edge and has $|E|$ triangles, a stabilizing node can only achieve this many triangles if it connects to nodes in the same copy of G from the three different possible copies. This means that we can label nodes in G^* depending on which stabilizing nodes they connect to. The stabilizing nodes are required to ensure the three copies of G as subgraphs of G^*, without these nodes, it would be possible to construct realizations where the copies intertwine and G^* would not have the copies of G.

Given the coloring part $\mathcal{P}_c$, we assign a color to c^1, c^2, c^3, then we assign a color for each $v \in V^1$ according to their connections to the coloring nodes. The coloring is valid since coloring nodes do not form any triangles ($t_{c^i} = 0$). □

There are heuristics and local search methods (MCMC [33], simulated annealing [57], tabu search [75]) that can achieve clustering coefficients close to target in many scenarios. In this dissertation, we do not evaluate NPM with local clustering coefficients due to the practical considerations of running an MCMC from realizations with low clustering as shown in [33].

Directed NPM (DNMP)

We provide a definition to extend NPM to represent directed graphs by their in and out degrees of nodes separately, which we refer to as *Directed NPM (DNPM)*.

Definition 4.3. *Given a directed graph $G(V, E)$ and two partitions $\mathcal{P}_{in}$, $\mathcal{P}_{out}$ with d_1 and d_2 parts, DNPM is a $|V| \times (d_1 + d_2)$ matrix, such that if $j < d_1$, $NPM[i, j]$ counts the number of edges coming into node i from nodes in part j of $\mathcal{P}_{out}$; or $NPM[i, j]$ counts the number of edges going from node i to nodes in part j of $\mathcal{P}_{in}$, if $j \geq d_1$.*

Similar to NPM and Theorem 4.1, DNPM is decomposable into directed degree sequence problems or equivalently into bipartite degree sequence problems with non-chords (to avoid

self-loops) as described in Section 1.3. Realizability and graph construction can be done efficiently based on directed degree sequence results [29]. The space of realizations is connected for simple DNPM realizations using double edge swaps and additional triangular C_6 swaps [25].

In the above definition $\mathcal{P}_{in}$ and $\mathcal{P}_{out}$ can be the same partition. The two partitions for in and out degrees allow for more flexibility in graph construction, this concept is related how D2K was modeled in [67]. DNPM always preserves D1K (directed degree sequence) and with the right partition D2K.

NPM with non-chords

Above, we elaborated on directed NPM realizations, which impose a special case of non-chord constraints. However, we observe that the realizability and graph construction problems for NPM with arbitrary non-chords can be solved by applying Tutte's gadget [70] for the degree sequence problems and sampling is possible using methods developed by Jerrum, Sinclair, Vigoda [46] or a simplified method for degree sequences with non-chords in [26] to get approximate random realizations. Details are omitted due to lack of space.

4.3.3 Tasks and Metrics

In Section 4.3.1, we described how to answer the realizability question and provided a construction algorithm for NPM. Here, we describe (i) the different tasks NPM can be used for and (ii) the corresponding evaluation metrics, appropriate for each task; these metrics will be assessed in the Evaluation section (4.4). Next, we describe the relation of NPM to classic graph construction problems and the setup for evaluating NPM in this context. Then, we describe how NPM can be used as a graph embedding for graph reconstruction,

link prediction and node classification tasks. A schematic example is provided for each task in Fig. 4.1. For each task, the input is a real graph and the NPM computed from it based on a node partition $\mathcal{P}$.

Graph Construction

The main goal in classic graph construction problems and the dK-series is to construct an ensemble of graphs that resemble real-world graphs. This is done by targeting certain properties such as degree sequences or joint degree matrices and observing similarity on a set of other properties such as eigenvalues, clustering coefficients, shortest path distributions for the constructed realizations. NPM by definition targets the following properties: the degree sequence of a graph ($\sum_{j=0}^{d} NPM[i,j]$ is the degree of node i), the partition-labeled two path distribution and the mixing matrix according to the partition or PAM. If the partition of nodes is by degree, then NPM preserves 2K; and the degree-labeled two paths which is part of 3K in dK-series terminology.

We have described in Section 4.3.1 how to sample NPM realizations for this task using an MCMC approach.

Graph Reconstruction

The main goal of this task is to exactly reconstruct a graph given an NPM input computed from the same graph. We evaluate the correctness of the reconstruction by the following metrics as defined in [38]:

Precision@k captures the correctness of the top k predicted edges, $E_{pred}(1:k)$:

$$Precision@k = \frac{|E_{pred}(1:k) \cap E_{obs}|}{k}, \tag{4.1}$$

where $E_{obs} = E$ for graph reconstruction, *i.e.* the input graph.

MAP estimates the precision for every node and computes the average over all nodes, as follows:

$$MAP = \sum_{i \in V} \frac{\sum_k Precision@k(i) \cdot \mathbb{I}\{E_{pred_i}(k) \in E_{obs_i}\}}{|\{k : E_{pred_i}(k) \in E_{obs_i}\}|} / |V| \tag{4.2}$$

Here,

$$Precision@k(i) = \frac{|E_{pred_i}(1:k) \cap E_{obs_i}|}{k}, \tag{4.3}$$

and E_{pred_i} and E_{obs_i} are the predicted and observed edges for node i.

NPM has the limitation that edges in every realization are 0-1 choices, *i.e.* an edge exists or an edge does not exist. To differentiate between edges for top-k prediction, our approach is to use the empirical probability for each edge occurring over multiple realizations. We approximate these probabilities by using MCMC sampling algorithm to generate multiple pseudo-random instances and measure the empirical probabilities for every edge to occur. Edges that are constrained by the NPM are going to have probability 1 (they must occur in every realization), while edges that have low probability to occur are the ones that are less constrained by the NPM.

Link Prediction

The main goal of link prediction is to suggest new edges that are not observed in the input graph. This evaluation is performed with a train and test split of edges of the real graph, then compute an NPM input on the training edges and evaluate predicted edges against the test edges using Precision@k and MAP, where E_{obs} is set to the test edges. We use empirical edge probabilities for top k recommendations as described for graph reconstruction.

Node Classification

The main goal of this task is to use the embedding vectors in NPM for each node as a feature vector for a node classification. Nodes are labeled with some attributes (*e.g.* gender in social networks). The NPM is computed for the input real graph and a classifier is trained on a training set of nodes and we use following metrics as defined in [38] to evaluate the classifier on a test set:

Macro-F1, in a multi-label classification task, is the average F1-score of all labels, *i.e.*

$$\textit{macro-F1} = \frac{\sum_{l \in L} F1(l)}{L}, \tag{4.4}$$

where $F1(l)$ is the F1-score for label l and L is the set of labels.

Micro-F1 calculates $F1$ globally by counting the total true positives, false negatives and false positives, giving equal weight to each instance. It is defined as follows:

$$\textit{micro-F1} = \frac{2 \cdot P \cdot R}{P + R}, \ \textit{where } P = \frac{\sum_{l \in L} TP(l)}{\sum_{l \in L}(TP(l) + FP(l))}; R = \frac{\sum_{l \in L} TP(l)}{\sum_{l \in L}(TP(l) + FN(l))} \tag{4.5}$$

are precision (P) and recall (R) respectively. $TP(l)$, $FP(l)$ and $FN(l)$ denote the number of true positives, false positives and false negatives, respectively, among the instances associated with the label l.

4.4 Evaluation

4.4.1 Experiment Setup

In this section, we quantitatively evaluate NPM, following the pipeline depicted in Fig. 4.1. We find that it outperforms graph embedding baselines on the graph construction task, while it performs comparably to baselines on embedding tasks (graph reconstruction, link prediction, node classification).

Datasets. Table 4.1 provides an overview of the publicly available datasets used in our experiments. We focus on undirected graphs without non-chords. As a pre-processing step, we remove any self-loops and isolates to ensure realizability of the targeted NPM. In Table 4.1, we also report the time averaged over 20 runs required to sequentially construct pseudo random realizations of NPM based on degree partition, *i.e.* these times include the time to run an MCMC sampler as well. Pseudo random NPM realizations were constructed in 1435 seconds for the Youtube graph used in [38] with 1,138,499 nodes and 2,990,443 edges. However, the GEM library [37] was not able to handle evaluation without falling back to subsampling nodes, for this reason we use smaller graphs.

We have executed our algorithms using Python 2.7 on a machine with an 48 core AMD Opteron 2.4Ghz processor. We have used NetworkX [40], igraph (Python version) [15], GEM [37] and node2vec [50].

Methods. We considered and evaluated several node partitions: random assignment, com-

Table 4.1: Datasets used in evaluation: number of nodes ($|V|$) and edges ($|E|$); average degree, number of (unique) degrees and labels (for node classification task), NPM construction time.

Name	Rice [69]	BlogCatalog [74]	Hep-Th [49]	Astro-ph [49]
$\|V\|$	4,087	10,312	9,875	18,771
$\|E\|$	184,828	333,983	25,973	198,050
Avg. degree	90.45	64.78	5.26	21.1
No. of degrees	348	576	55	234
No. of labels	3	39	-	-
Const. time	10sec	28sec	2sec	11sec

munity detection algorithms available from igraph [15] and degree-based partitions, with the heuristic applied to achieve exactly d parts. Here, we report results for the following partitions: degree sequence (*NPM_RND_1*); DSM or NPM with degree-based partition where $d =$ #unique degrees (values shown in Table 4.1) (*NPM_degree_d*), NPM with degree-based partition where $d = 128$ (*NPM_degree_128*), NPM with partition from leading eigenvectors-based community detection algorithm [56] where $d = 128$ (*NPM_eigen_128*). We also evaluate baseline embedding methods: *HOPE*, *Laplacian Eigenvectors (LAP)* and *node2vec* (with default parameters [37] and $d = 128$).

Tasks and Metrics. We consider the tasks of Section 4.3.3: the results for graph construction are presented in Section 4.4.2 and the results for the graph embedding tasks (graph reconstruction, link prediction, node classification) are presented in Section 4.4.3. The performance metrics are task-specific and have been described in detail in Section 4.3.3.

4.4.2 Graph Construction Task

In this section, we demonstrate how well NPM can target graph properties using the following setup: we read each input graph, assign a partition, compute the corresponding NPM input and construct 20 realizations using implementation based on NetworkX [40] combined with an MCMC sampler. In addition to NPM realizations, we construct realizations for

graph embedding methods that contain the top $|E|$ number of edges recommended from the embedding.

We present our results for all our test graphs and construction methods in Fig. 4.4. Due to lack of space, we omit results for several graph properties that we have evaluated, such as degree distributions and shortest paths, since the former is exactly targeted by any realization of NPM and the latter is mostly met by random graphs with the small-world property.

Since NPM preserves degree sequences, we expect it to capture degree correlations. While this is not true with $d = 1$, *i.e.* degree sequences, *NPM_eigen* captures a fair amount of the target degree correlations without specifically targeting it using $d = 128$, see Fig. 4.4(b). By definition, *NPM_degree* preserves degree correlations exactly when $d = \#$unique degrees. However, we also observe that using only $d = 128$ by merging parts of the partition is still a viable option even if it introduces some error as for Astro-Ph in Fig. 4.4(d). When the number of unique degrees is less than 128 and we split parts of the partition, we also preserve degree correlations exactly as shown for Hep-Th in Fig. 4.4(c).

NPM-based graph construction (depending on the partition method and d) targets the leading eigenvalues of the adjacency matrix reasonably well. This is interesting, since NPM does not directly target this property and in other related work for dK-series, eigenvalues are usually not well captured (after the leading eigenvalue) [57], [67]. This improvement could be because NPM uses node level information opposed to partition level information (such as 2K or D2K).

In Fig. 4.4, we observe that clustering is not well captured for most graphs (except BlogCatalog), this is expected for random realizations of NPMs consistently with what prior work has shown this phenomena in relation with the dK-series. As we mentioned in Section 4.3.2, this can be improved using heuristics and MCMC to better target clustering coefficients. However, we should note that NPM performs best for BlogCatalog in the next section, which

suggests that targeting clustering coefficients is a good direction.

Graph embedding methods perform poorly compared to NPM due to the many isolates in their realizations. This leads to significant differences for the observed metrics in this section. The performance varies among embedding methods but they are all outperformed by our proposed NPM method.

4.4.3 Graph Embeddings Tasks

Here, we demonstrate that NPM with using only 128 dimensions (*NPM_eigen_128* and *NPM_degree_128*) performs comparable to other baseline methods. We vary the ensemble size, *i.e.* the number of realizations generated (1 vs. 100) by NPM to produce edge probabilities for graph reconstruction and link prediction tasks as discussed in Section 4.3.3. These ensembles are not required for node classification (embedding vectors are directly used as features), hence we omit them from Table 4.3. We have used an 80%-20% train-test split for link prediction and node classification.

Table 4.2 shows that in terms of MAP, NPM performs comparable for graph reconstruction on several datasets, outperforms *node2vec* on all datasets but only outperforms *HOPE* for BlogCatalog for link prediction. Lower performance on link prediction could be due to that NPM only captures edges present in the training graph, but if no edges are present across partitions, it will never construct them during tests.

In terms of Precision@k, we observe that sampling more realizations boosts performance (across datasets) as shown in Fig. 4.5. This confirms our expectation from Section 4.3.1.

NPM outperforms all the evaluated graph embedding methods on the BlogCatalog graph (where clustering was close to real graph) for graph reconstruction and link prediction. In other datasets, NPM comes mostly second to *HOPE* in both tasks. We see similar

Table 4.2: MAP values for Graph reconstruction / Link prediction tasks for different methods ($d = 128$)

Name	Rice	BlogCatalog	Hep-Th	Astro-Ph
NPM_eigen 1	0.31/0.09	0.37/0.05	0.03/0.02	0.03/0.02
NPM_eigen 100	0.31/0.05	0.48/0.09	0.05/0.02	0.03/0.04
NPM_degree 1	0.34/0.04	0.33/0.05	0.30/0.00	0.15/0.01
NPM_degree 100	0.30/0.02	0.36/0.07	0.36/0.01	0.21/0.01
LAP	0.31/0.13	0.06/0.01	0.28/0.16	0.20/0.20
HOPE	0.46/0.30	0.46/0.04	0.15/0.10	0.15/0.16
node2vec	0.06/0.02	0.08/0.02	0.11/0.02	0.03/0.01

Table 4.3: Micro, Macro F1 values for Node classification task for different methods ($d = 128$)

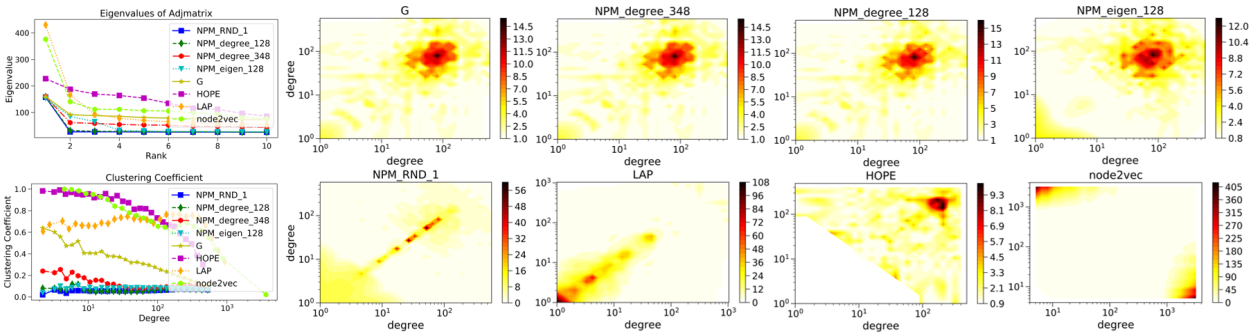
Name	Rice	BlogCatalog
NPM_eigen	0.572, 0.392	0.236, 0.103
NPM_degree	0.573, 0.391	0.217, 0.096
LAP	0.555, 0.345	0.194, 0.048
HOPE	0.559, 0.356	0.186, 0.042
node2vec	0.597, 0.466	0.370, 0.200

performance for node classification across different methods (in this case *node2vec* dominates) but NPM-based methods slightly outperform *LAP* and *HOPE*.

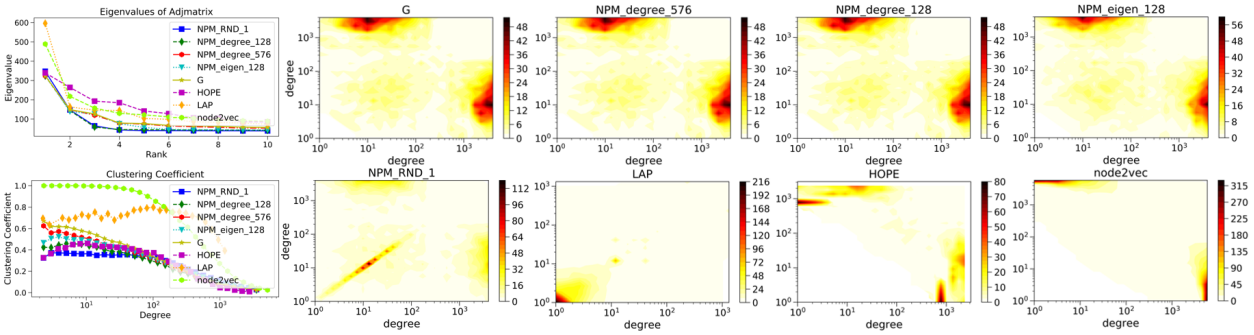
4.5 Summary

We have proposed a novel way of looking at graph construction methods using node neighborhoods. Our model, Neighborhood Partition Matrix (NPM) generalizes previous work in graph construction, by extending the notion of degree groups intro arbitrary parts, and includes them as special cases [9], [8]. NPM can be used as a flexible framework, by choosing different node partitions. When used as a graph embedding, NPM has the following advantages: (1) the embedding dimensions become more interpretable and (2) there can be deterministic guarantees on local graph properties (such as degree sequence, degree correlation) - with the right choice of partition. We have also proposed extensions of NPM to include properties such as clustering coefficients and DNPM for directed graphs. NPM's

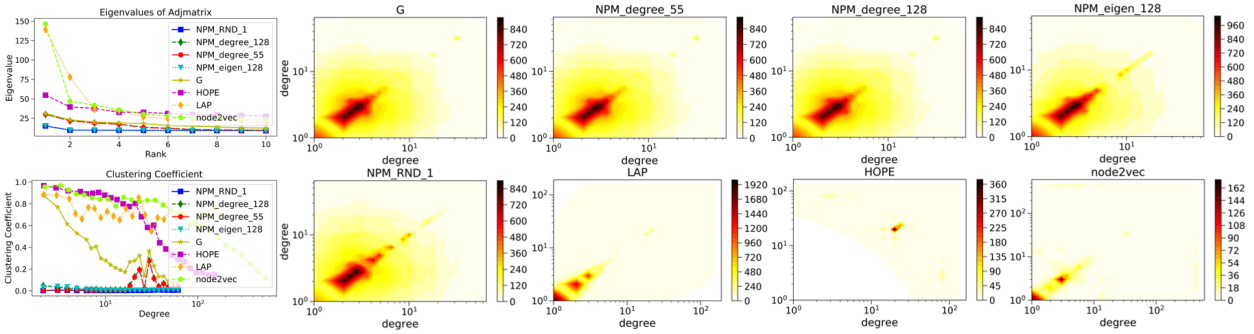
performance is superior or – at worst– comparable to baseline graph embedding methods, while NPM has the previously mentioned qualitative improvements for graph construction.



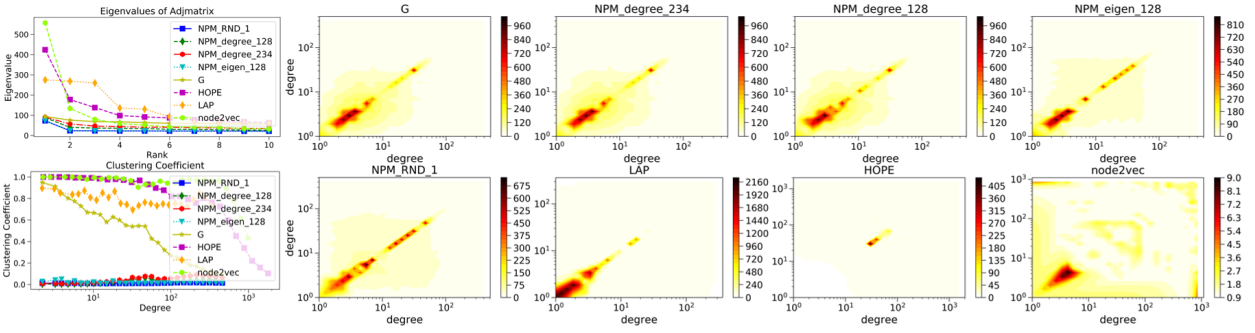
(a) The real graph, G, is Rice.



(b) The real graph, G, is BlogCatalog.



(c) The real graph, G, is Hep-Th.



(d) The real graph, G, is Astro-Ph.

Figure 4.4: **Graph Construction Task.** Evaluation of various NPM models and graph embeddings on how well they match three real graphs, G, w.r.t. various graph construction metrics (clustering coefficients, leading eigenvalues, degree correlations).

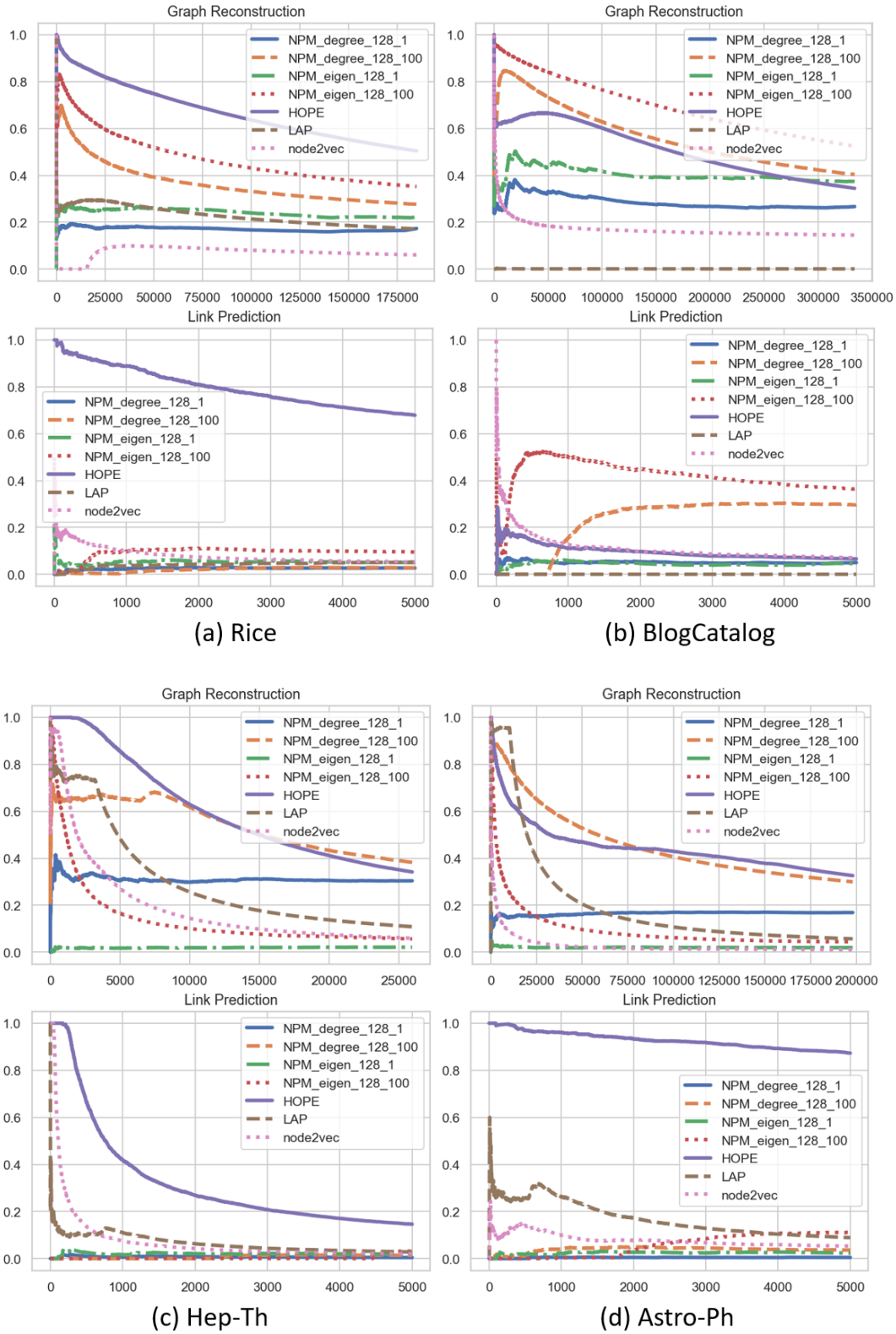


Figure 4.5: Precision@k for top-k recommendations. Top subfigures: **Graph Reconstruction Task** (up to $k = |E|$). Bottom subfigures: **Link Prediction** (up to $k = 5000$).

Chapter 5

Conclusion

In this dissertation, we have developed algorithms that construct synthetic graphs that resemble real-world graphs. This work builds on and extends a large body of literature on generating realizations of undirected graphs in terms of well-defined structural properties, such as a degree sequence or a joint degree matrix.

Our 2K+ framework advances the state-of-the-art in modeling and simulation of complex networks, especially in the context of online social networks that exhibit high clustering and are affected by node attributes. It provides an efficient way to construct simple, undirected, directed and directed acyclic graphs, that exhibit exactly a target degree correlation and potentially additional properties, including: clustering, number of connected components, node attributes for undirected and average neighbor degree, number of mutual edges or balanced realizations for directed graphs, etc. Key strengths of this work include: (1) a principled approach to graph construction, with exact guarantees when possible (2K, 2K+A, 2K+CC, D2K) and efficient heuristics when necessary (*e.g.* the 2.25K, 2.5K and 3K+ problems are NP-hard, thus motivating heuristics); (2) extensibility to target additional properties by exploiting the insights we developed, namely the under-defined nature of the

2K algorithm (*e.g.* order of adding edges), manipulating attributes in JDAM, speeding up MCMC, and connections between all these related problems; (3) efficiency: the time for constructing large graphs, in practice, reduced from weeks and days to minutes and seconds. We have also contributed our implementations to the Python NetworkX library, both for undirected [32] and for directed [65] graphs.

We have also proposed a novel way of looking at graph construction methods using node neighborhoods partitioned into d parts. This notion generalizes degree groups and our model, Neighborhood Partition Matrix (NPM), includes previous work as special cases [9], [8]. In addition, NPM can be used as a framework by assigning different partitions of nodes in graphs. In the context of graph embeddings, NPM has several interesting properties, namely (1) the embedding dimensions become more interpretable and (2) there can be strong guarantees on local graph properties (such as degree sequence, degree correlation) - with the right choice of partition. We have also discussed extensions of NPM with global properties such as clustering coefficients and DNPM for directed graphs. NPM's performance is comparable to other baseline graph embedding methods while NPM has the previously mentioned qualitative improvements for graph construction.

Bibliography

[1] Simple Graph: http://mathworld.wolfram.com/SimpleGraph.html.

[2] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A. J. Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd international conference on World Wide Web*, pages 37–48. ACM, 2013.

[3] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 171–180. Acm, 2000.

[4] G. Amanatidis, B. Green, and M. Mihail. Graphic realizations of joint-degree matrices. *arXiv preprint arXiv:1509.07076*, 2015.

[5] G. Amanatidis, B. Green, and M. Mihail. Connected realizations of joint-degree matrices. *Discrete Applied Mathematics*, 2018.

[6] C. Bachmaier, A. Gleißner, and A. Hofmeier. Dagmar: Library for dags. *Department of*, 2012.

[7] D. A. Bader and V. Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. Technical report, Georgia Institute of Technology, 2006.

[8] M. D. Barrus and E. A. Donovan. Neighborhood degree lists of graphs. *Discrete Mathematics*, 341(1):175–183, 2018.

[9] K. E. Bassler, C. I. Del Genio, P. L. Erdős, I. Miklós, and Z. Toroczkai. Exact sampling of graphs with prescribed degree correlations. *New Journal of Physics*, 17(8):083052, 2015.

[10] O. Bastert and C. Matuszewski. Layered drawings of digraphs. In *Drawing graphs*, pages 87–120. Springer, 2001.

[11] M. Belkin and P. Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Advances in neural information processing systems*, pages 585–591, 2002.

[12] J. Blitzstein and P. Diaconis. A sequential importance sampling algorithm for generating random graphs with prescribed degrees. *Internet Mathematics*, 6(4):489–522, 2011.

[13] C. Carstens. A uniform random graph model for directed acyclic networks and its effect on motif-finding. *Journal of Complex Networks*, 2(4):419–430, 2014.

[14] C. R. Coullard and P. H. Ng. Totally unimodular leontief directed hypergraphs. *Linear algebra and its applications*, 230:101–125, 1995.

[15] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[16] É. Czabarka, A. Dutle, P. L. Erdős, and I. Miklós. On realizations of a joint degree matrix. *Discrete Applied Mathematics*, 181:283–288, 2015.

[17] E. Czabarka, L. A. Szekely, Z. Toroczkai, and S. Walker. An algebraic monte-carlo algorithm for the bipartite partition adjacency matrix realization problem. *arXiv preprint arXiv:1708.08242*, 2017.

[18] C. I. Del Genio, H. Kim, Z. Toroczkai, and K. E. Bassler. Efficient and exact sampling of simple graphs with given arbitrary degree sequence. *PloS one*, 5(4):e10012, 2010.

[19] W. Devanny, D. Eppstein, and B. Tillman. The computational hardness of dk-series. In *NetSci 2016*, 2016.

[20] N. Developers. Networkx. *networkx. lanl. gov*, 2010.

[21] X. Dimitropoulos, D. Krioukov, A. Vahdat, and G. Riley. Graph annotations in modeling complex network topologies. *ACM Trans. Model. Comput. Simul.*, 19(4):17:1–17:29, Nov. 2009.

[22] S. Dorogovtsev. Networks with desired correlations. *arXiv preprint cond-mat/0308336*, 2003.

[23] P. Erdős and T. Gallai. Gráfok előírt fokú pontokkal. *Mat. Lapok*, 11:264–274, 1960.

[24] P. L. Erdős, S. G. Hartke, L. van Iersel, and I. Miklós. Graph realizations constrained by skeleton graphs. *arXiv preprint arXiv:1508.00542*, 2015.

[25] P. L. Erdős, Z. Király, and I. Miklós. On the swap-distances of different realizations of a graphical degree sequence. *Combinatorics, Probability and Computing*, 22(03):366–383, 2013.

[26] P. L. Erdős, S. Z. Kiss, I. Miklós, and L. Soukup. Constructing, sampling and counting graphical realizations of restricted degree sequences. *arXiv preprint arXiv:1301.7523*, 2013.

[27] P. L. Erdos, I. Miklós, and Z. Toroczkai. A decomposition based proof for fast mixing of a markov chain over balanced realizations of a joint degree matrix. *SIAM Journal on Discrete Mathematics*, 29(1):481–499, 2015.

[28] P. L. Erdős, I. Miklós, and Z. Toroczkai. New classes of degree sequences with fast mixing swap markov chain sampling. *arXiv preprint arXiv:1601.08224*, 2016.

[29] D. R. Fulkerson et al. Zero-one matrices with zero trace. *Pacific J. Math*, 10(3):831–836, 1960.

[30] D. Gale et al. A theorem on flows in networks. *Pacific J. Math*, 7(2):1073–1082, 1957.

[31] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical computer science*, 1(3):237–267, 1976.

[32] M. Gjoka. 2k simple implementation in networkx. `https://networkx.github.io/documentation/latest/reference/generated/networkx.generators.joint_degree_seq.joint_degree_graph.html`, 2016.

[33] M. Gjoka, M. Kurant, and A. Markopoulou. 2.5 k-graphs: from sampling to generation. In *INFOCOM, 2013 Proceedings IEEE*, pages 1968–1976. IEEE, 2013.

[34] M. Gjoka, B. Tillman, and A. Markopoulou. Construction of simple graphs with a target joint degree matrix and beyond. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1553–1561. IEEE, 2015.

[35] M. Gjoka, B. Tillman, A. Markopoulou, and R. Pagh. Efficient construction of 2k+ graphs. In *NetSci 2014*, 2014.

[36] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

[37] P. Goyal and E. Ferrara. Gem: A python package for graph embedding methods. *Journal of Open Source Software*, 3(29):876.

[38] P. Goyal and E. Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 2018.

[39] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[40] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, Aug. 2008.

[41] S. L. Hakimi. On realizability of a set of integers as degrees of the vertices of a linear graph. i. *Journal of the Society for Industrial and Applied Mathematics*, 10(3):496–506, 1962.

[42] V. Havel. Poznámka o existenci konecnych grafu. *Časopis pro pěstování matematiky*, 80(4):477–480, 1955.

[43] P. W. Holland and S. Leinhardt. Local structure in social networks. *Sociological methodology*, 7:1–45, 1976.

[44] P. Holme and B. J. Kim. Growing scale-free networks with tunable clustering. *Physical review E*, 65(2):026107, 2002.

[45] D. R. Hunter, M. Handcock, C. Butts, S. M. Goodreau, and M. Morris. ergm: A package to fit, simulate and diagnose exponential-family models for networks. *Journal of Statistical Software*, 24(3), 2008.

[46] M. Jerrum, A. Sinclair, and E. Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *Journal of the ACM (JACM)*, 51(4):671–697, 2004.

[47] B. Karrer and M. E. Newman. Random graph models for directed acyclic networks. *Physical Review E*, 80(4):046110, 2009.

[48] H. Kim, C. I. Del Genio, K. E. Bassler, and Z. Toroczkai. Constructing and sampling directed graphs with given degree sequences. *New Journal of Physics*, 14(2):023012, 2012.

[49] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[50] J. Leskovec and R. Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[51] F. Maffray and M. Preissmann. On the np-completeness of the k-colorability problem for triangle-free graphs. *Discrete Mathematics*, 162(1):313–317, 1996.

[52] P. Mahadevan, D. Krioukov, K. Fall, and A. Vahdat. Systematic topology analysis and generation using degree correlations. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 135–146. ACM, 2006.

[53] S. Micali and V. V. Vazirani. An $o(\sqrt{|V|}|e|)$ algorithm for finding maximum matching in general graphs. In *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pages 17–27. IEEE, 1980.

[54] M. Mihail. On generating graphs with prescribed degree sequences for complex network modeling applications. In *3rd Workshop on Approximation and Randomization Algorithms in Communication Networks, 2002*, 2002.

[55] A. Mislove, M. Marcon, K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, 2007.

[56] M. E. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical review E*, 74(3):036104, 2006.

[57] C. Orsini, M. M. Dankulov, P. Colomer-de Simón, A. Jamakovic, P. Mahadevan, A. Vahdat, K. E. Bassler, Z. Toroczkai, M. Boguñá, G. Caldarelli, et al. Quantifying randomness in real networks. *Nature communications*, 6, 2015.

[58] M. Ou, P. Cui, J. Pei, Z. Zhang, and W. Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114. ACM, 2016.

[59] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[60] J. J. Pfeiffer III, S. Moreno, T. La Fond, J. Neville, and B. Gallagher. Attributed graph models: modeling network structure with correlated attributes. In *Proc. of WWW*, 2014.

[61] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, 290(5500):2323–2326, 2000.

[62] T. A. B. Snijders, P. E. Pattison, G. L. Robins, and M. S. Handcock. New specifications for exponential random graph models. *Sociological Methodology*, 36:99–154, 2006.

[63] I. Stanton and A. Pinar. Constructing and sampling graphs with a prescribed joint degree distribution. *Journal of Experimental Algorithmics (JEA)*, 17:3–5, 2012.

[64] R. Taylor. *Constrained switchings in graphs*. University of Melbourne, Department of Mathematics, 1980.

[65] B. Tillman. D2k simple implementation in networkx. `https://github.com/networkx/networkx/pull/3551`, 2019.

[66] B. Tillman and A. Markopoulou. On the number of connected components of joint degree matrix realizations. In *abstract submitted to NetSci 2018*, 2018.

[67] B. Tillman, A. Markopoulou, C. T. Butts, and M. Gjoka. Construction of directed 2k graphs. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1115–1124, New York, NY, USA, 2017. ACM.

[68] B. Tillman, A. Markopoulou, M. Gjoka, and C. T. Butts. 2k+ graph construction framework: Targeting joint degree matrix and beyond. *IEEE/ACM Transactions on Networking*, 27(2):591–606, April 2019.

[69] A. Traud, P. Mucha, and M. Porter. Social Structure of Facebook Networks. *Arxiv preprint arXiv:1102.2166*, 2011.

[70] W. T. Tutte. A short proof of the factor theorem for finite graphs. *Canad. J. Math*, 6(1954):347–352, 1954.

[71] F. Viger and M. Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. In *International Computing and Combinatorics Conference*, pages 440–449. Springer, 2005.

[72] B. Viswanath, A. Mislove, M. Cha, and K. Gummadi. On the evolution of user interaction in facebook. In *Proc. WOSN*, 2009.

[73] Y. Amanatidis and B. Green and M. Mihail. Graphic realizations of joint-degree matrices. *Unpublished manuscript*, 2008.

[74] R. Zafarani and H. Liu. Social computing data repository at ASU, 2009.

[75] Y. Zhu, Y.-R. Song, and Y.-W. Li. A tabu search optimization algorithm with 2.5 k null model. In *2018 37th Chinese Control Conference (CCC)*, pages 1082–1086. IEEE, 2018.