Karlsruhe Institute of Technology

An Efficient Generator for Large Clustered Dynamic Random Networks

Bachelor Thesis of

Roland Kluge

At the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	Prof. Dr. rer. nat. Dorothea Wagner
Advisor:	Dipl.-Inform. Andrea Schumm
Second advisor:	Dr. rer. nat. Robert Görke

Duration: 1 May 2011 – 5 August 2011

Danksagung

Meinen beiden Betreuern, Andrea Schumm und Robert Görke, gilt mein Dank dafür, dass sie mir bei weit mehr als einer schwierigen Frage mit hilfreichen Hinweisen zur Seite gestanden haben. Obwohl ihr Zeitplan stets gefüllt war, konnte ich mich jederzeit an sie wenden, wenn es offene Punkte zu besprechen gab. Das Thema der Arbeit wurde von ihnen ausgeschrieben und gefiel mir von der ersten Sekunde an.

Abschließend möchte ich meinen Eltern und meiner Freundin für die Mühen danken, die sie mit dem Korrekturlesen und den zahlreichen Hinweisen auf sich genommen haben. Was ihren Beitrag noch wertvoller macht, ist die Tatsache, dass sie alle in Bereichen tätig sind, die fachlich gesehen weit von der theoretischen Informatik entfernt liegen.

Selbständigkeitserklärung

Hiermit versichere ich, dass die vorliegende Arbeit sowie der veröffentlichte Quelltext meine eigene Arbeit sind. Arbeit anderer wurde durch Quellenverweise oder Nennung des Autors (Quelltext) eindeutig kenntlich gemacht.

Statement of Authorship

I hereby declare that this document and the accompanying code have been composed by myself and describe my own work, unless otherwise acknowledged in the text. It has not been accepted in any previous application for a degree.

Karlsruhe, 5 August 2011

Zusammenfassung

Dynamische Clusteralgorithmen sind ein hochgradig aktuelles Forschungsthema. Die immer wichtiger werdenden sozialen Netzwerke sind ein anschauliches Beispiel für dynamische und geclusterte Strukturen:

Mitglieder sozialer Netzwerke bilden oft Gemeinschaften, die sich im Laufe der Zeit in Größe und Anzahl verändern. Die eigentliche Beobachtungsgröße hingegen sind oft einzig und alleine die Teilnehmer und deren Beziehungen untereinander. Deren Wegfallen oder Entstehen sowie die Fluktuation von Mitgliedern führen dazu, dass sich die Struktur der Gemeinschaft mit der Zeit verändert.

Dynamische Clusteralgorithmen dienen dazu, diese sich wandelnden Strukturen zu erkennen, um weitere Auswertungsschritte (etwa gezielte Werbung oder auch zu Planungszwecken) zu erleichtern. Allerdings stehen Referenzdaten für derartige Szenarien leider nicht in dem Umfang zur Verfügung, wie es zur Evaluierung von dynamischen Clusteralgorithmen wünschenswert und nötig wäre.

Diese Bachelorarbeit stellt die theoretische Konzeption und Implementierung eines Testdatengenerators vor, mit dessen Hilfe eine zeitlich und speichertechnisch effiziente Erzeugung von Zufallsgraphen möglich ist, die sowohl geclustert als auch dynamisch sind. Die Leistung dieser Arbeit liegt darin, einen bereits existierenden Generator abzulösen, der zwar zeitlich, nicht aber speichertechnisch hinreichend leistungsfähig ist, um große Netzwerke zu erzeugen.

Abstract

Algorithms for dynamic and clustered scenarios are of topical importance in the clustering research community. Social networks which are illustrative real-world examples of dynamic and clustered structures perpetually gain in importance.

Members of these networks constitute communities which evolve in size and number. However, communities may not be directly recognizable for an external observer, for example if it comes to informal substructures of larger communities. In these situations, the only measurable indicator are the members' relationships among each other. Over time the network evolves as new acquaintances come to life and old ones are lost. Some members may even leave the community, cutting all ties at once.

Dynamic clustering algorithms serve for detecting such communities in order to allow further processing which may for example be personalized advertisement or for planning purposes. Unfortunately, suitable reference data for testing dynamic clustering algorithms is difficult to find and varies in quality.

This thesis describes the theoretical design and practical implementation of a generator producing dynamic, clustered, and random networks which is both space and time efficient. It's fundamental achievement is the thorough redevelopment of a previous generator resulting in an implementation with dramatically smaller memory consumption and improved running time.

Contents

1. Introduction

1.1. Motivation

The efficient analysis of large collections of data is one of today's great challenges. Whenever the amount of data exceeds a certain level it is not affordable to take every single data set into consideration. In general, we are more interested in how far certain subsets of the acquired data belong together in a logical, semantical or functional sense. An established approach is to group data sets into so-called *clusters* and observe the resulting, more coarse-grained structure.

We frequently encounter clustered data in almost all fields of science and everyday life: Virtual and real social networks are divided into communities; market research strives towards categorizing customers into groups in order to make them favorable offers; networks of collaboration of actors, politicians or scientists[1] can be sources of inspiration for new theoretical models (see Section 1.4.2).

The broad variety of different types of data can often be mapped to one of the most versatile tools computer scientists know: *graphs*. Entities are mapped to nodes and the observed relationships to edges. However, having translated the data into a graph, it is not so clear how to subdivide the data sets into clusters. In fact, no single answer to this question exists but a broad variety of approaches has been proposed in the past. Fortunato has published an overview of the most widely used clustering techniques which also contains an appealing collection of illustrative real-world examples for clustered data (see Section 2 in [10]).

Frequently, the networks under consideration evolve over time. Coming back to the example of social networks, we observe that communities may split due to diverging interests and other communities may coalesce if there are new common interests or projects on which the members of the participating communities work together.

Even though real-world dynamic clustered instances with a reference clustering exist – that is, we know the clustering of the data in advance – there are several reasons why we are interested in generating artificial test data: First, we would like to produce graphs with a set of predefined properties such as the distribution of node degrees or size of the clusters, enabling us to examine the behavior on dynamic clustering algorithms under almost arbitrary circumstances. Furthermore, large real-world instances are not numerous and, in general, they are subject to certain restrictions which prohibit the independent comparison of clustering algorithms or implementations thereof.[2]

[1]The Erdős number is popular in this context [15].

[2]Some years ago, similar problems existed in the route planning community where real-world road networks generally were subject to nondisclosure agreements. The online service OpenStreetMap [11] and

Christian Staudt and Robert Görke implemented a random graph generator for the described clustered and dynamic scenario. Their generator performs well in practice but the size of the generated instances is limited by the memory consumption of the generator which is constantly quadratic in the number of nodes. Realistic instances, however, are sparse graphs, that is, the edge count is linear in the number of nodes. The objective of this work is now to design and implement a generator which is able to produce significantly larger test data instances than the preceding implementation, enabling the developers of dynamic clustering algorithms to test the behavior with networks being of comparable size as large real-world instances. Therefore, formally speaking, we want the new generator to consume at most $\mathcal{O}(n+m)$ memory, where n is the number of nodes and m is the number of edges, in comparison to $\mathcal{O}(n^2)$ before.

1.2. Overview

The thesis is structured as follows: Chapter 1 is a collection of all information that is fundamental for understanding the theory of the generator. Therefore, in Section 1.3, we first clarify our notion of the terms and symbols appearing in this thesis. Based on these definitions, Section 1.4 formalizes the scenario described colloquially above. Afterwards, we take a closer look at the *source and target tree* approach being the theoretical concept behind the previous generator in Section 1.5.

Having learnt about all necessary basics we present our *cluster tree* approach in Chapter 2, which is the theoretical part of this thesis. Of course, we need to clarify first what exactly caused the inadequate memory consumption of the existing generator. In the course of this chapter we completely redesign the internal structure of the generator starting with the indexing scheme for nodes and edges, proceeding with edge and node operations and finishing with cluster operations.

Chapter 3, will describe the actual implementation. The implementation notes in Section 3.1 are not a design document but can be rather read as a user manual comprising the command line parameters and being followed by examples and compatibility issues concerning the former generator. Section 3.4 closes the chapter with a small set of experiments giving a sense of the generator's performance.

Chapter 4 closes this work. The summary in Section 4.1 states whether we were able to meet all requirements concerning space and time efficiency of the new generator. As in every theoretical work answers to previous questions yield new questions. Some of them are listed in Section 4.2.

Finally, we mention that a nomenclature can be found at the very end of this work which contains all symbols appearing in this thesis.

1.3. Preliminaries

1.3.1. General Terms

Given a set S and an element e we abbreviate the operation $S' = S \setminus \{e\}$ with $S' = S - e$ and $S' = S \cup \{e\}$ with $S' = S + e$.

For two sets A and B the *unordered Cartesian product* is defined as the set of all unordered pairs with exactly one element from A and B: $A * B = \big\{\{a,b\} \mid a \in A, b \in B\big\}$.

For a given set S the set $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ is called a partitioning of S if the following two properties hold:

- $\mathcal{S}$ covers S: $\bigcup_{i=1}^{n} S_i = S$.
- The S_i are disjoint: $\forall i, j = 1 \ldots n, i \neq j : S_i \cap S_j = \emptyset$.

the steadily growing community around it finally redeemed the researchers from this limitation.

Suppose there exists a set of elements S and a function $p : S \to [0..1]$. The *probability mass* of S weighted with p is defined as: $P(S,p) = \sum_{e \in S} p(e)$.

Given a discrete random variable $X : S \to \mathbb{N}$ with S being the set of all possible *outcomes*. The probability that X takes value x_i is denoted with $P[X{=}x_i]$ and the *expectation value* of X is $\mathbb{E}[X] = \sum_{x_i \in S} x_i \cdot P[X{=}x_i]$.

The *conditional probability* $P[X{=}x_i|Y{=}y_j]$ is the probability that random variable X takes value x_i under the condition that Y is known to have value y_i. Mathematically it is defined as: $P[X{=}x_i|Y{=}y_j] = \frac{P[X=x_i, Y=y_j]}{P[Y=y_j]}$.

The *binomial coefficient* $\binom{n}{k}$ is the number of subsets containing k elements which can be chosen from n distinct elements: $\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$.

The symbol $\perp$ symbolizes an invalid pointer (*null* pointer). For example, it may be returned by a hash table when there exists no entry for a given key. Setting an entry to $\perp$ is equivalent to deleting this entry from the data structure.

1.3.2. Bachmann-Landau Notation

For given functions $f, g : D \to \mathbb{N}$ we say that

- f grows *asymptotically not faster than* g (for short: $f \in \mathcal{O}(g)$) if

$$\exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

- f grows *asymptotically as fast as* g (for short: $f \in \Theta(g)$) if

$$\exists c_1, c_2 \in \mathbb{R}, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

1.3.3. Graphs

An *undirected graph* is defined as an ordered pair $G = (V, E)$, where V is the set of *nodes* and E is the set of *edges*. An edge $e = \{u, v\} \in V * V$ is an unordered pair of nodes from V. The nodes u and v are called endpoints of e and e *connects* u and v. If $\{u, v\} \notin E$, u and v are *unconnected*. For $\{u, v\} \in E$ we say that u and v are *adjacent* (shortly: $u \sim v$) and that e is *incident* to u and to v (shortly: $e \sim u, v$). The neighborhood of node v is the set of all adjacent nodes of v: $N(v) = \{u \in V \mid u \sim v\}$.

The definition of E implies two properties of G: First, G does not contain edges which have the same node as twofold endpoint, so-called *loops*. Second, there is at most one edge connecting two distinct nodes in G. Graphs having these two properties are called *simple*. If not otherwise stated, the size of the sets V and E is denoted with $n = |V|$ and $m = |E|$, respectively.

The *degree* of a node v is the number of nodes being adjacent to v: $\deg v = |N(v)|$. If $\deg v = 0$ then v is called *isolated*.

A simple graph is called *complete* if each node has degree $n-1$, implying that $m = \binom{n}{2}$. Given a set V of nodes, the set of all possible undirected edges between nodes of V is denoted with $\binom{V}{2} = V * V$.

For a given graph G its *complement graph* $\bar{G}$ is defined as $\bar{G} = (\bar{V} = V, \bar{E} = \binom{V}{2} \setminus E)$, that is, $\bar{G}$ embodies the same set of nodes and all possible edges except for those which exist in G. The pairs of nodes in $\bar{E}$ are called *non-edges* of G and $\bar{m} := \bar{E}$.

Given a subset $V' \subseteq V$. The *node induced subgraph* $G(V') = (V', E')$ of G embodies all edges between nodes of V': $E(V') = E \cap (V' * V')$.

1.3.4. Clusters

A *clustering* $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ is a partitioning of V where each of the C_i is non-empty implying that $|\mathcal{C}| \in \mathcal{O}(|V|)$. If not defined otherwise, the variable k refers to the number of clusters in $\mathcal{C}$: $k := |\mathcal{C}|$. The function $c : V \to \mathcal{C}$ maps each node v to its cluster C: $c(v) = C :\Leftrightarrow v \in C$.

For a cluster C the graph $G(C) = (C, E(C))$ is the node induced subgraph of C, where $E(C)$ are the *intra-cluster edges* of C. For $e \in E(C)$ we use the abbreviation $e \in C$. $m(C)$ is the number of edges in $G(C)$ and $\bar{m}(C) = \left|\binom{V(C)}{2} \setminus E(C)\right|$ is the number of *intra-cluster non-edges* of C. Edges having endpoints in two distinct clusters are called *inter-cluster edges*. In a clustered graph each edge is either an inter-cluster or intra-cluster edge. A pair of nodes $\{u, v\}$ is called *intra-cluster pair* or *inter-cluster pair* if $c(u) = c(v)$ or $c(u) \neq c(v)$, respectively.

Let $v \in C_i$ where $C_i \in \mathcal{C}$ is a cluster. The *intra-cluster neighborhood* $N_{\text{intra}}(v)$ of v is the set of all nodes u in C_i which are adjacent to v: $N_{\text{intra}}(v) = \{u \sim v \mid c(u) = c(v)\}$. Similarly, the *inter-cluster neighborhood* $N_{\text{inter}}(v)$ of v is: $N_{\text{inter}}(v) = \{u \sim v \mid c(u) \neq c(v)\}$.

Intra-cluster degree and *inter-cluster degree* of v are defined as the size of the respective neighborhood: $\deg_{\text{intra}}(v) = |N_{\text{intra}}(v)|$ and $\deg_{\text{inter}}(v) = |N_{\text{inter}}(v)|$. Therefore, we see that $N(v) = N_{\text{intra}}(v) \cup N_{\text{inter}}(v)$ and $\deg v = \deg_{\text{intra}}(v) + \deg_{\text{inter}}(v)$.

If there exists only a single cluster or if each cluster contains only one node, that is $|\mathcal{C}| \in \{1, n\}$, $\mathcal{C}$ is called a *trivial clustering* of G.

The set of all *intra-cluster edges* is called $E_{\text{intra}} = \bigcup_{i=1}^{k} E(C_i)$ and $m_{\text{intra}} = |E_{\text{intra}}|$. Conversely, the set of all *intra-cluster non-edges* is called $\bar{E}_{\text{intra}} = \bigcup_{i=1}^{k} \left(\binom{C_i}{2} \setminus E(C_i)\right)$ where $\bar{m}_{\text{intra}} = |\bar{E}_{\text{intra}}|$.

For *inter-cluster edges* we introduce $E_{\text{inter}} = E \setminus E_{\text{intra}}$ and $m_{\text{inter}} = |E_{\text{inter}}|$. The set of all *inter-cluster non-edges* is defined as $\bar{E}_{\text{inter}} = \bar{E} \setminus \bar{E}_{\text{intra}}$ the count of which is abbreviated with $\bar{m}_{\text{inter}} = |\bar{E}_{\text{inter}}|$.

Given two clusters C_i, C_j. The set of *inter-cluster edges* between C_i and C_j is $E(C_i, C_j) = \{u \sim v \mid u \in C_i, v \in C_j\}$ and $m(C_i, C_j) := |E(C_i, C_j)|$. The set of *inter-cluster non-edges* is defined as $\bar{E}(C_i, C_j) = (C_i * C_j) \setminus E(C_i, C_j)$ and $\bar{m}(C_i, C_j) := |\bar{E}(C_i, C_j)|$.

1.3.5. Graph and Clustering Adjacency

The graphs $G = (V, E), G' = (V', E')$ are *adjacent* (shortly: $G \sim G'$) if any one of the following *graph operations* transforms G into G'

- node insertion: $V' = V + v, v \notin V$
- node deletion: $V' = V - v, v \in V$
- edge insertion: $E' = E + e, e \in \bar{E}$
- edge deletion: $E' = E - e, e \in E$

Note that adjacency is a symmetric relation.

We define the following *cluster operations*:

- merge two clusters: $\mathcal{C}' = \left(\mathcal{C} - C_x - C_y\right) + C'_z$ where $C_x, C_y \in \mathcal{C}$ and $C'_z = C_x \cup C_y$
- split a cluster: $\mathcal{C}' = \left(\mathcal{C} - C_x\right) + C'_y + C'_z$ where $C_x \in \mathcal{C}$ and $C_x = C'_y \cup C'_z$

When clusters C_1, C_2 are merged into one cluster C_3, the operation is abbreviated with $(C_1, C_2) \to C_3$. Conversely, for the case of splitting cluster C_1 into C_2 and C_3 we write: $C_1 \to (C_2, C_3)$.

1.4. The Dynamic, Clustered, and Random Scenario

We now possess an adequate vocabulary for describing dynamic, clustered random graphs. The following sections introduce some basics in the field of clustering and random graph models. Towards the end of this section we will formalize the graph model (Section 1.4.3) and give a theoretical definition of a random graph generator (Section 1.4.4).

1.4.1. Indices for the Quality of a Clustering

Over time several criteria have been suggested which could be used to evaluate whether for a graph $G = (V, E)$ the partitioning $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$ of V is a good clustering.

Coverage

Intuitively, one might consider a clustering $\mathcal{C}$ of G to be "good" if the number of intra-cluster edges contributes a large fraction of all edges of G. This ratio is called *coverage* [4] of $\mathcal{C}$:

$$\mathrm{cov}(\mathcal{C}) = \frac{m_{\text{intra}}}{m}$$

Unfortunately, an obvious drawback is observable: If $|\mathcal{C}| = 1$ the resulting coverage is 1 and if $|\mathcal{C}| = n$ we obtain $\mathrm{cov}(\mathcal{C}) = 0$. Thus neither the maximum nor the minimum value for the coverage can be taken as measure of quality for the clustering.

Performance

The *coverage* approach focuses on the number of existing edges but it does not prune a clustering if it is sparse, that is, if there are also many intra-cluster non-edges. The index *performance* [19] takes this into account: The sum of intra-cluster edges and inter-cluster non-edges is related to the maximum number of edges.

$$\mathrm{perf}(\mathcal{C}) = \frac{m_{\text{intra}} + \bar{m}_{\text{inter}}}{\binom{n}{2}}$$

This measure prunes clusterings which are relatively coarse and it favors clusterings which possess many intra-cluster edges and only a few inter-cluster edges.

Intra-Cluster Density vs. Inter-Cluster Sparsity

Another criterion for deciding on the quality of a given clustering is the so-called *intra-cluster density vs. inter-cluster sparsity* paradigm.

No formalized, commonly accepted definition of this criterion exists, so that we give a rather intuitive one: Roughly speaking, the paradigm states that $C \subseteq V$ is a cluster, if the fraction of the number of existing intra-cluster edges to the number of all possible intra-cluster edges is much larger than the fraction of the number of existing inter-cluster edges to the number of all possible inter-cluster edges.

To make this clearer, we assume that for a certain node $v \in C$ two values d and s exist, where

$$d = \frac{\deg_{\text{intra}}}{n(C) - 1}$$
$$s = \frac{\deg_{\text{inter}}}{n - n(C)}$$

According to the *density vs. sparsity* paradigm, d must in average be (much) larger than s in order to recognize C as a cluster. Note that this is not an exact definition but shall summarize the rather long description above.

Other indices

We only present the unweighted version of the quality criteria as weighting is of no importance in our scenario. Görke additionally summarizes definitions for weighted graphs [14]. A wide range of other quality indices for graph clusterings exist. Common measures are described in [5]. For the sake of completeness we mention modularity [17] and inter-cluster conductance [6].

1.4.2. Random Graph Models

Random graph models can be used for characterizing the results of a random graph generation process. Such models state certain probabilistic properties of the generated graphs which can be as ordinary as "Each graph contains exactly n nodes and m edges" but there also exist non-trivial statements, for example about the average node degree or the expected maximal size and count of connected components.

The meaning of the terms *process* and *graph model* is sometimes not clearly separated: A process takes parameters and generates a graph whereas a model points out properties of a certain class of graphs which are generated by a process. Therefore, many algorithms generating graphs of the same model may exist.

Some graph models bear a descriptive name and a concise one which only emphasizes the necessary parameters, for example Gilbert's model is a synonym for $\mathcal{G}(n,p)$; this model takes two parameters n and p. For small-world networks and preferential attachment model, the authors of the original publications did not define a concise name so that we introduced it for clarifying what parameters the specific model takes.

Gilbert's Model $\mathcal{G}(n,p)$

Gilbert's model [13], also called $\mathcal{G}(n,p)$, is a widely used model for creating random graphs with n nodes[3]. For two nodes u and v the probability that edge $e = \{u,v\}$ exists in a graph G generated with $\mathcal{G}(n,p)$ is p. The expected degree of a node v is $\mathbb{E}[\deg v] = p \cdot (n-1)$.

A way to generate graphs using $\mathcal{G}(n,p)$ is to initialize the n nodes and, afterwards, to draw for all $\binom{n}{2}$ edges whether each of them exists or not. Clearly, this takes time $\mathcal{O}(n^2)$. Batagelj and Brandes showed how to dramatically reduce the running time to $\mathcal{O}(n+m)$, where m is the number of edges created (see Section II.A. in [2]).

Erdős-Rényi Model $\mathcal{G}(n,m)$

Generators adhering to the Erdős-Rényi model [7] which is also called $\mathcal{G}(n,m)$ generate graphs with exactly n nodes and m edges. Not considering isomorphisms, each graph fulfilling these constraints is equally likely to be created.

A naive approach for generating a graph with $\mathcal{G}(n,m)$ is to draw m edges from the set of all possible $\binom{V}{2}$ edges. If the drawn edge already exists, another one is tried until a non-edge is found. Clearly, the expected running time of a single edge selection increases in the course of the process because drawing an existing edge becomes increasingly probable. Batagelj and Brandes suggest two efficient implementations of $\mathcal{G}(n,m)$ and present experimental results [2]. In the course of discussing the second implementation they introduce the *virtual Fisher-Yates shuffle*, a data structure which allows for at most two trials per drawn edge. As we will see in Section 2.2 this data structure may also prove useful in our work.

Small-World Networks

The *small-world phenomenon* first stated by Stanlay Milgram [16] has emerged to great popularity not only in science but also in public. A similar description of this phenomenon is the "six degrees of separation": According to this theory, most people on the world are connected with each other via at most six acquaintances. For some communities such as

[3]Some publications also claim that both models, $\mathcal{G}(n,p)$ and $\mathcal{G}(n,m)$, are called Erdős-Rényi model.

mathematicians (Erdős number [15]) and actors (“six degrees of Kevin Bacon”), this theory has been analyzed on the basis of collaboration graphs.

In graph theory Watts and Strogatz proposed an approach on how to generate so-called small-world networks [20]. In principle the generation process of a small-world network deterministically initializes a graph G_0 with n nodes which are spread regularly over a circle and each node is connected to its k closest neighbors. The distance to a neighbor is measured in hops which have to be performed on the circumference in order to reach that neighbor.

The process now iterates k times over all nodes in a (counter-)clockwise order and decides for each node v whether and, if yes, in how far each incident edge $e = \{u, v\}$ is *rewired*, that is whether the adjacent node u shall be substituted with another node u'. If edge $\{v, u'\}$ already exists, e remains untouched. p is the probability that an edge is rewired.

Watts and Strogatz use two measures for characterizing the resulting graphs: The first one is the characteristic path length $L(p)$ which is the average minimal distance of any two nodes. The second one is the clustering coefficient $C(p)$ which is the average of the completeness of the neighborhood $N(v)$ of each node v. The completeness of $N(v)$ is the number of existing edges in the node induced subgraph $G(N(v)) = (N(v), E')$ related to the maximum number of edges within $G(N(v))$: $C(p) = \text{avg}_{v \in V} \left({}^{|E'|}/_{\binom{|N(v)|}{2}} \right)$. The key observation is that the characteristic path length $L(p)$ sharply decreases if p differs slightly from 0 and that $C(p)$ stays almost constant for the respective values of p.

Characteristic path length $L(p)$ and completeness of the neighborhood $C(p)$ can be seen as the characteristic properties of the small-world graph model $\mathcal{G}(L, C)$.

In a real-world scenario, where the nodes represent persons and edges acquaintances, this yields a small “degree of separation” (low $L(p)$) while the community structure is preserved (high $C(p)$). Following these observations this type of graphs is called small-world network.

Preferential Attachment

Gilbert’s model, the Erdős-Rényi model and small-world model share one property: The degree of most nodes is quite close to the arithmetic mean in all three models.

A random graph model which does not share this property is called *preferential attachment* and has been first published by Barabási and Albert [1] and refined by Bollobás et al. [3].

This random model follows a *power law* and it can be abbreviated as $\mathcal{G}(n, \gamma)$. Formally speaking, the probability of node v having k neighbors adheres to $P[\deg v = \nu] \propto \nu^{-\gamma}$ for some constant γ, that is the probability of high degrees ν decreases exponentially in ν.

The motivation of this model bases upon an observation in self-organizing networks such as social networks: If a new person joins the network and if he/she may choose to establish a certain amount of relationships with other members, he/she will probably choose one or more of those (few) participants that already maintain a relatively large amount of relations – and therefore are “important” in some way.

A graph consistent with the preferential attachment model can be produced by the following strategy: Nodes v are gradually added to the existing network and are immediately connected to k neighbors. The probability that a node u is chosen as new adjacency of v is proportional to $\deg u$.

1.4.3. The Model $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$

Our objective is to create clusterings that are detectable using the paradigm of *intra-cluster density vs. inter-cluster sparsity.* A suitable random graph model for this criterion is $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$; it is derived directly from $\mathcal{G}(n, p)$ and has been proposed in [5] and [12].

The parameter n of the model denotes the fixed number of nodes. p_{in} and p_{out} are the edge probabilities for intra-cluster and inter-cluster edges, respectively. p_{out} is a single value, whereas p_{in} is a list of size k, the cluster count: $p_{\text{in}} = \big(p_{\text{in}}(C_1), \ldots, p_{\text{in}}(C_k)\big)$. For two nodes u and v the probability for edge $e = \{u, v\}$ to exist in a graph created with $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$ is called *edge probability* $p(u, v)$ (also: $p(e)$) where

$$p(u,v) = \begin{cases} p_{\text{in}}(C_i) & u, v \in C_i \\ p_{\text{out}} & \text{otherwise} \end{cases}$$

Note that the concept of edge probabilities is similar to the definition of the *sparsity vs. density* paradigm. We call the clustering $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$ the *ground truth* because converging towards this clustering is the long-term objective of all graph operations.

Node Degrees

As for Gilbert's model we describe here how we can use the edge probabilities to calculate the expected intra- and inter-cluster node degrees. The expected intra-cluster node degree $\deg_{\text{intra}}(C_i)$ within cluster C_i can be found when considering the generation of $G(C)$ as Gilbert's model with parameters $n(C_i)$ and $p_{\text{in}}(C_i)$ and we obtain:

$$\mathbb{E}[\deg_{\text{intra}}(C_i)] = p_{\text{in}} \cdot (n(C_i) - 1).$$

For the inter-cluster node degree $\deg_{\text{inter}}$, the definition is slightly different: As all nodes $v \in C_i$ within cluster C_i have uniform expected $\deg_{\text{inter}}(v)$, this value is abbreviated with $\deg_{\text{inter}}(C_i)$. The process of establishing inter-cluster links of v is similar to the one used for creating the intra-cluster adjacencies. Iterating over all nodes $u \in V \setminus C_i$ outside of C_i, we create an edge $\{u, v\}$ with probability p_{out}.

$$\begin{aligned}
\mathbb{E}[\deg_{\text{inter}}(C_i)] &= p_{\text{out}} \cdot (n - n(C_i)) \\
\mathbb{E}[\deg_{\text{inter}}] &= \mathbb{E}[\frac{1}{n} \sum_{v \in V} \deg_{\text{inter}}(v)] \\
&= \frac{1}{n} \sum_{v \in V} \mathbb{E}[\deg_{\text{inter}}(v)] \\
&= \frac{1}{n} \sum_{v \in V} p_{\text{out}} \cdot (n - n(c(v))) \\
&= \frac{1}{n} \sum_{i=1}^{k} n(C_i) \cdot p_{\text{out}}(n - n(C_i)) \\
&= \frac{p_{\text{out}}}{n} \sum_{i=1}^{k} n(C_i)(n - n(C_i))
\end{aligned}$$

The probabilities p_{out} and p_{in} should be chosen such that $p_{\text{out}} \ll p_{\text{in}}(C_i)$ because the possible number of inter-cluster adjacencies of a node v is much larger than the number of possible intra-cluster adjacencies. Therefore, a change in p_{out} has a larger impact on the inter-cluster node degree than does a similar change to one of the p_{in} values.

Cluster Sizes

The sizes of the k clusters are not defined by the input parameters of the $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$ model. A sensible suggestion made in [12] and [5] is to choose a node's cluster uniformly at random which entails a binomial distribution of the cluster sizes around the mean value $\frac{n}{k}$.

Furthermore we may introduce a coefficient β which *skews* the binomial distribution as follows: Each cluster C_i is assigned to an equidistant subinterval $\left[\frac{i-1}{k}, \frac{i}{k}\right)$ of $[0, 1)$. When searching for a cluster to add a new node to we draw an integer $i \in [0, k-1]$. Now, we add the node to the cluster which is assigned to the surrounding interval of $\left(\frac{i}{k}\right)^\beta$ where $\beta = 1$ in the unskewed case. If $\beta > 1$ then the result of the exponentiation is always smaller than the base which causes clusters C_i with relatively small indices i to receive more nodes in comparison to those with relatively indices.

1.4.4. Dynamics

Up to now we did neither include time dependency into our scenario nor did we clarify what we exactly talk about when using the term generator.

Interface of the Generator

A generator of dynamic clustered random graphs is an algorithm which receives the following parameters:

- n: number of nodes of the initial instance
- $p_{\text{in}} = \{p_{\text{in}}(C_1), \ldots, p_{\text{in}}(C_k)\}$, implying cluster count k
- p_{out}: inter-cluster edge probability
- $t_{\max}$: number of time steps
- p_ω: probability of a cluster operation
- p_μ: probability of a *split* operation ($1 - p_\mu$: probability of a *merge* operation)
- p_χ: probability of an edge operation ($1 - p_\chi$: probability of a node operation)
- p_ν: probability of adding a node ($1 - p_\nu$: probability of removing a node)

The output of the algorithm is a list of pairs: $\mathcal{S} = (G_t, \mathcal{R}_t)$, indexed with the time steps $t = 0 \ldots t_{\max}$ where

- $G_t = (V_t, E_t)$ is a graph
- $\mathcal{R}_t$ is the clustering which is considered to be recognizable by an observer (*reference clustering*)

and G_0 is called the *initial instance*. Finally, we claim that the G_t are adjacent: $\forall t = 1 \ldots T : G_t \sim G_{t-1}$, that is, exactly one node or edge is added or deleted per time step. Note that deleting a node entails deleting its adjacencies and thus a node operation normally implies several edge operations.

Internals Structure of the Generator

Figure 1.1 depicts the detailed generation process. For all further considerations we adhere to this diagram so that it may provide for useful orientation. Robert Görke and Christian Staudt provided the diagram.

We see that there is one procedure which has not been explained, yet: $\text{ws}(P_E, P_{\bar{E}})$. This function is called a *weighted selection* (see Algorithm 1). It takes two weights and non-deterministically determines whether the element associated with the first weight is to be selected. Otherwise, the second element is to be chosen. The probability of selecting a specific element is proportional to its associated weight compared to the sum of both weights.

In our generator the weighted selection is responsible of steering the edge count towards the expected value: P_E and $P_{\bar{E}}$ are the probability masses of E weighted with $1 - p(u,v), \{u,v\} \in E$ and $\bar{E}$ weighted with $p(u,v), \{u,v\} \in \bar{E}$:

$$P_E = \sum_{\{u,v\} \in E} (1 - p(u,v)) \tag{1.4.1}$$

$$P_{\bar{E}} = \sum_{\{u,v\} \in \bar{E}} p(u,v) \tag{1.4.2}$$

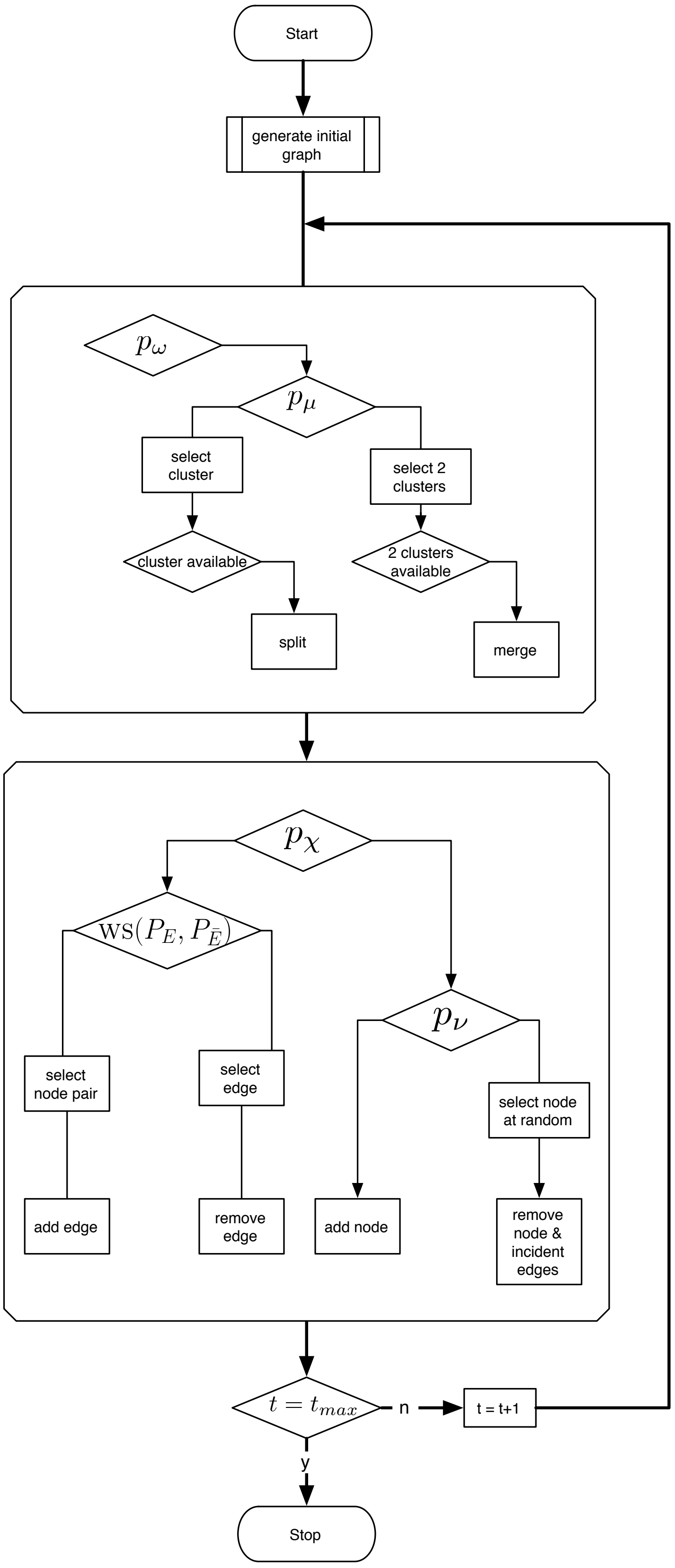


Figure 1.1.: Decision tree of the dynamic graph generator: *Rhombi* are decisions and *rectangles* are subprocedure calls.

Algorithm 1: Weighted selection among two elements

```
Input     : $\omega_1$,$\omega_2$: weights of the two elements
Output    : $c$: whether to select the first element
// Draw real number uniformly at random from the interval $[0, \omega_1 + \omega_2)$
$r \leftarrow$ randomReal $(\omega_1 + \omega_2)$
if $r < \omega_1$ then
    $c \leftarrow$ true
else
    $c \leftarrow$ false
```

Both probability masses are equal in the expected case and edge insertions are as probable as edge deletions. If there exist fewer edges than expected the probability mass of all non-edges $P_{\bar{E}}$ is larger than P_E and the generator tends to insert an edge rather than removing one.

The subsequent paragraphs deal with open questions which may arise from defining the generator in the above way. Most of the points have quite a fine-grained scope but they have to be mentioned anyway. For the first reading, the reader may skip directly to Section 1.5.4 where we summarize the resulting requirements which the generator has to meet.

Selecting Clusters for Cluster Operations

One question is whether we are restricted in the selection of clusters for *split* and *merge* operations. In fact, we are not: Our model makes no assumptions about the size distribution of clusters and, consequently, each cluster may be selected with uniform probability. The only restriction we put on the clusters for a new cluster operation is that the do not want the clusters to be already participating in another operation. This restriction allows us to clearly separate the operations from one another. As we defined clusters to be non-empty, a cluster participating in a *split* operation additionally has to contain at least two nodes.

Calculating new Values for p_{in}

When we trigger a *merge* or *split* operation we have to calculate p_{in}values for the resulting cluster(s). We could use the following strategy:

- For a *merge* operation $(C_1, C_2) \to C_3$ we define the new intra-cluster edge probability to be the arithmetic mean of $p_{\text{in}}(C_1)$ and $p_{\text{in}}(C_2)$: $p_{\text{in}}(C_3) := \frac{1}{2}\big(p_{\text{in}}(C_1) + p_{\text{in}}(C_2)\big)$.
- For a *split* operation $C_1 \to (C_2, C_3)$ the clusters C_2 and C_3 inherit the old value of p_{in}: $p_{\text{in}}(C_2) := p_{\text{in}}(C_3) := p_{\text{in}}(C_1)$.

However, these rules lead to an unfavorable steady equalization of the p_{in} values. Görke and Staudt suggested to use a Gaussian distribution $N(\mu, \sigma^2)$ to draw the next $p_{\text{in}}$where μ and σ are the arithmetic mean and the standard deviation of the original p_{in}values (see "Splitting and Merging Clusters" in [18]).

Reference Clustering

We have seen that the generation process is divided into time steps which are again subdivided into two parts: In the first part, the ground truth clustering of the graph is (potentially) modified without affecting the edges and nodes of the graph. An outside observer, who is only aware of edges and nodes, can by no means detect this change in the clustering. For this reason we introduce the *reference clustering* $\mathcal{R}$ which is the clustering which we deem recognizable by a clustering algorithm.

When performing a cluster operation the results will be written *immediately* into the ground truth but we postpone applying the operation to the reference clustering until we assume that the results may be seen by the clustering algorithm.

Completeness of Cluster Operations

We need to clarify when and how we determine whether a certain operation is considered to be complete and can be applied to the reference clustering. We may check for completeness after every graph operation or after each iteration and we use the observed inter-cluster edge count between the initial (*merge*) or resulting (*split*) clusters of the operation. As each cluster takes part in at most one operation at a time, we may measure the completeness of each running cluster operation separately. Our decision is based on a predefined *threshold* θ for the completeness.

For the operations $(C_1, C_2) \to C_3$ (*merge*) and $C_3 \to (C_1, C_2)$ (*split*), we define the constants a and b as follows:

$$\begin{aligned} a &:= n(C_1) \cdot n(C_2) \cdot p_{\text{out}} \\ b &:= n(C_1) \cdot n(C_2) \cdot p_{\text{in}}(C_3) \end{aligned}$$

where a is the expected number of inter-cluster edges between clusters C_1 and C_2 when C_1 and C_2 constitute different clusters and b is the expected number of intra-cluster edges between C_1 and C_2 when both are subsets of cluster C_3.

In each time step we count the number of edges $m(C_1, C_2)$ between C_1 and C_2 and determine if the evolution of the original cluster(s) to the desired cluster(s) has made enough progress in order to consider the operation to be complete. At the beginning of a *merge* operation, the expected value of $m(C_1, C_2)$ is $a = n(C_1) \cdot n(C_2) \cdot p_{\text{out}}$ because C_1 and C_2 are different clusters. C_1 and C_2 coalesce and therefore the generation process now pushes $m(C_1, C_2)$ towards $b = n(C_1) \cdot n(C_2) \cdot p_{\text{in}}(C_3)$. For a *split* operation, the generator will decrease $m(C_1, C_2)$ over time converging towards value a.

$$\text{complete}((C_1, C_2) \to C_3) = \begin{cases} \text{true} & \text{if } m(C_1, C_2) \geq \theta \cdot b + (1 - \theta) \cdot a \\ \text{false} & \text{otherwise} \end{cases}$$

$$\text{complete}(C_3 \to (C_1, C_2)) = \begin{cases} \text{true} & \text{if } m(C_1, C_2) \leq \theta \cdot a + (1 - \theta) \cdot b \\ \text{false} & \text{otherwise} \end{cases}$$

The tuning parameter θ determines how strict the decision is: If $\theta = 1$ then the operation is only considered complete if $m(C_1, C_2)$ perfectly matches or exceeds the expected value. Vice versa, for $\theta = 0$, the operation almost will immediately be considered to be complete in the expected case. It needs to be mentioned that choosing θ to be 1 is not sensible at all as it may take a long time for the edge count to *exactly* match the expected value.

Several cluster operations can finish at the same time. Note that, theoretically, a new cluster operation can be initiated in each time step. Indeed, the probability p_ω which steers the frequency of cluster operations is relatively small in real use cases.

Selecting Nodes and Edges for Graph Operations

We have to determine how to select a certain edge or node for graph operations. When it comes to removing a node, we argue that nodes shall be chosen uniformly at random from V. Note that none of the prerequisites of the $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$ model prevents us from proceeding in this way. Edges shall be selected with a probability proportional to $p(u, v)$ (*insert*) and $1 - p(u, v)$ (*remove*), where u, v are the endpoints of the edge under consideration. This follows directly from the definition of p_{in} and p_{out} (see Section 1.4.3).

1.5. Source and Target Tree Approach

This section presents the original approach for implementing the subprocedures in the decision tree depicted in Figure 1.1. This approach has been published in two documents:

[18] and Section 4.2 of [14]. For the sake of simplicity we will always refer to the former publication. The whole approach is based upon a data structure similar to an ordinary binary tree which is described first before we proceed to the actual implementation of the graph and cluster operations.

1.5.1. Randomized Binary Selection Tree

Contrary to the introduced convention, n denotes the number of elements stored in the tree within this section.

An augmented binary selection tree stores elements from a weighted set $S = \{(e, \omega)_i\}$ where $\omega \in \mathbb{R}$ is the weight associated with element e. For each element e we create a *tree node* $o_e = (e, \omega, \ell, r)$, where ℓ and r are pointers to the left and right child nodes, respectively. Leaf nodes have $\ell = r = \perp$. We assume that there exists an implicit parent pointer parent(o) and height $h(o)$ for each tree node. For the root tree node r we define: parent$(r) = r$ and $h(r) = 0$. Figure 1.2 illustrates the structure of the tree. The tree

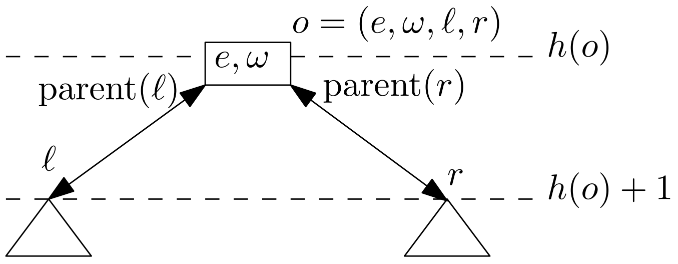


Figure 1.2.: Schematic view of a tree node $o = (e, \omega_e, \ell, r)$

offers element *insertion*, *deletion*, *weighted selection*, and *weight update*. Each node has an accumulated weight which is defined as follows:

Definition 1 (Accumulated Weight)**.** *The accumulated weight of tree node $o = (e, \omega, \ell, r)$ is defined as:*

$$\text{weight}(o) = \begin{cases} 0 & o = \perp \\ \omega + \text{weight}(\ell) + \text{weight}(r) & \textit{otherwise} \end{cases}$$

We precalculate weight(o) for all nodes o and update it if the tree changes. Essentially, the binary tree is an ordinary weighted binary tree which has been augmented with an accumulated weight. Like a binary tree the data structure still has size $\mathcal{O}(n)$. We will see that the costs for *insert* and *delete* are not affected by the augmentation.

Find Operation

Contrary to binary *search* trees the proposed binary tree is not used for deterministically retrieving certain elements. Therefore, we need not implement a *find* operation. We either access tree nodes by means of their associated element (e. g. when we have to update the weight of a tree node) or the tree supplies us with a pointer to a certain node (e. g. when we randomly select a tree node).

Selection

Algorithm 2 implements the *select* operation for the binary selection tree: We choose a real number d in the interval of $[0, \text{weight}(\text{root}))$ and descend the tree always making sure that we are in the leftmost subtree the weight of which is larger than d. It is for this reason that we need to correct d when descending into the right subtree. When we finally arrive at a node for which we find weight$(\ell) \leq d <$ weight$(\ell) + \omega$ we have arrived at the desired node and return it.

Algorithm 2: *select* operation for the binary selection tree

```
Input  : $\rho$: the root of the tree, $\Delta$: a real number in $[0, \text{weight}(\rho))$
Output : $e$: the selected element
$o = (e, \omega, \ell, r) \leftarrow \rho$
while $\Delta \notin [\text{weight}(\ell), \text{weight}(\ell) + \omega)$ do
    if $\Delta < \text{weight}(\ell)$ then
        $o \leftarrow \ell$
    else
        $o \leftarrow r$
        $\Delta \leftarrow \Delta - \text{weight}(\ell) - \text{weight}(r)$
```

Weight Update

The weight ω of an element e may change and we need to update the tree, recalculating the accumulated weight for all nodes on the path from o to the root r. As the length of all root to node paths is in $\mathcal{O}(\log n)$, the update operations for node o takes time $\Theta(h(o)) \subseteq \mathcal{O}(\log n)$. Algorithm 3 implements the *update* operation.

Algorithm 3: *update* operation for the binary selection tree

```
Input  : $e$: element, $\omega_{\text{new}}$: new weight
$o = (e, \omega_{\text{old}}, \ell, r)$
$p = \text{parent}(o)$
$\Delta \leftarrow \omega_{\text{new}} - \omega_{\text{old}}$
// $\text{parent}(p) = p \Leftrightarrow p$ is root
repeat
    $o.\omega \leftarrow o.\omega + \Delta$
    $o \leftarrow p$
    $p \leftarrow \text{parent}(o)$
until $p = o$
```

Insertion

Inserting an element (e, ω) into the tree, we create a new tree node $o = (e, \omega, \perp, \perp)$. A tree node p with minimal height $h(p)$ and at least one unassigned child pointer becomes the parent of o: $\text{parent}(o) \leftarrow p$.

Of course, we have to run the *update* operation with parameters o and $\text{weight}(o) = \omega$. The running time for the *insert* operations is $\mathcal{O}(\log n)$.

Deletion

Deleting an existing tree node o from the tree a leaf node b with maximal height $h(b)$ is selected and replaces o in the tree, that is, all pointers of o are copied to b and the child pointer in $\text{parent}(o)$ is replaced with b.

Afterwards, the *update* operation is run for b and the former parent of b. The running time for the *delete* operations is $\mathcal{O}(\log n)$.

Example: *select* operation

After describing all operations the binary tree supports we give a small example of a *select* operation in Figure 1.3. The figure illustrates the last but one step during the *select* procedure. Each edge is labeled with the accumulated weight of the subtree below it and each tree node is labeled with its weight ω.

- We start at root node $\textcircled{1}$ and choose an offset in the interval $[0, 5.4)$: In this case $\Delta = 3.8$.

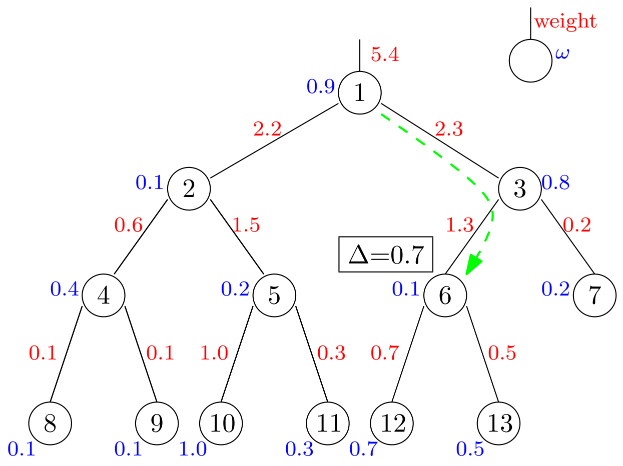


Figure 1.3.: Possible choice for initial offset Δ which leads to a selection of node 6

- The first comparison yields false but for the second comparison we find $3.8 \geq 3.1 = 2.2 + 0.9$ and therefore we descend into the *right* subtree adjusting the offset to: $\Delta = 3.8 - 3.1 = 0.7$.

- Arriving now at node $\textcircled{3}$ it holds that $0.7 < 1.3$ and we descend into the *left* subtree not modifying the offset: $\Delta = 0.7$.

- The current situation is depicted in Figure 1.3: We have now reached node $\textcircled{6}$ and it neither holds that $0.7 < 0.7$ nor that $0.7 \geq 0.8$. For this reason, the current node is the result of the *select* operation.

1.5.2. Source and Target Trees

The binary selection tree is the only data structure which is required in order to understand Görke's and Staudt's approach. We will see that we can perform all graph and cluster operations using a certain set of augmented binary trees.

The data structure consists of so-called source and target trees which are divided into data structures for the *insert* operation and for the *delete* operation. We describe the data structure for deleting edges first and we will see that the corresponding one for inserting edges can be defined analogously: We need to be able to choose from the set of edges E. There exists one *source-tree of deletion* T containing all nodes from V and each node u is additionally associated with a *target tree of deletion* $T(u)$ representing the neighborhood of u. Therewith all edges are covered twice by the target trees as illustrated in Figure 1.4.

The weights for nodes in the source tree and target trees of insertion are defined as follows:

$$\omega(u) = \sum_{v \in N(u)} (1 - p(u,v)) \qquad \text{weight in } T \qquad (1.5.1)$$

$$\omega_u(v) = 1 - p(u,v) \qquad \text{weight in } T(u) \qquad (1.5.2)$$

We see that the tree nodes in the source tree are weighted with the probability mass of all weights in the respective target tree and notice the similarity of (1.5.1) and (1.4.1).

Having defined the source and target trees for insertion we see that we can implement the data structure for inserting edges in exactly the same way using *source and target trees of insertion* this time: The source tree of insertion $\bar{T}$ again contains the nodes of V and the target tree of insertion $\bar{T}(u)$ represents the neighborhood $\bar{N}(u)$ of u in $\bar{G}$ which

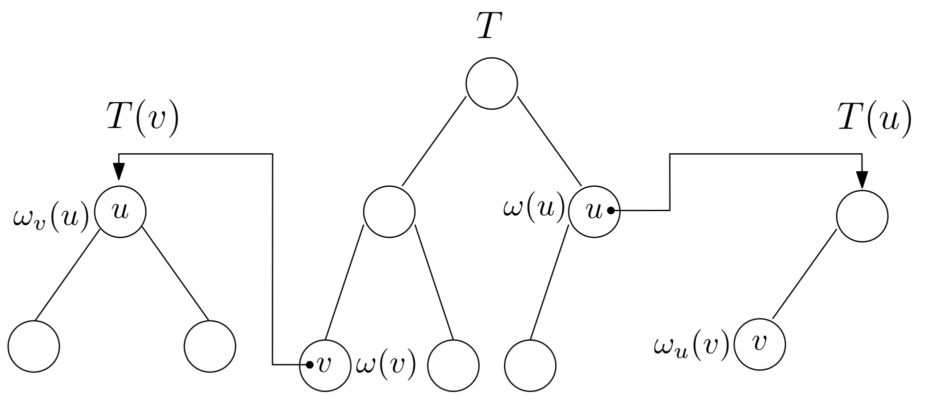


Figure 1.4.: Schematic view of the source and target trees for insertion: The *source tree of deletion* T represents a graph with 6 nodes. We obtain the size of the neighborhoods of u and v from their associated *target trees deletion*: $|N_u| = 2$ and $|N_v| = 3$.

contains all potential adjacencies of u which do not exist in G. For the weights of the tree nodes we find:

$$\bar{\omega}(u) = \sum_{v \in \bar{N}(u)} p(u, v) \qquad \text{weight in } \bar{T} \tag{1.5.3}$$

$$\bar{\omega}_u(v) = p(u, v) \qquad \text{weight in } \bar{T}(u) \tag{1.5.4}$$

and see that the probability mass (1.4.2) corresponds to (1.5.3).

Edge Deletion

Having defined the data structures selecting an edge for deletion is now straightforward: First we select a node u from the source tree of deletion and afterwards we choose a node v from the target tree of deletion $T(u)$.

The edge $e = \{u, v\}$ is deleted by removing v from the target tree of deletion $T(u)$ and by updating the weight of u in the source tree of deletion T. As we store each edge twice we have to run the described operations in an analogous way for source node v and target node u. Additionally, we have to make e available for selection in a future time step. Therefore, v is added to the target tree of insertion $\bar{T}(u)$ and the source tree of insertion $\bar{T}$ has to be updated. In order to create the second half of e we add u to $\bar{T}(v)$ and update the weight of the tree node belonging to v in the source tree.

Edge Insertion

Edge insertion is symmetric to edge deletion. The only difference is now, that we choose a non-edge from the source and target trees of *insertion*. All other operations are analogous.

Node Deletion

When deleting node v, we remove its target trees for deletion and insertion $T(v), \bar{T}(v)$ and its tree node in the source trees $T, \bar{T}$. Furthermore, for all nodes $u \in N(v)$ we remove v from the target tree of deletion T_u and for all nodes $u \in \bar{N}(v)$ we delete v from the target tree of insertion $\bar{T}(u)$ updating the weights of all affected nodes in both source trees. The operation runs in time $\Theta(n)$ if deleting a single tree node takes constant time because the number of elements in $T(v)$ and $\bar{T}(v)$ always sums up to n.

Node Insertion

Inserting node v, we add a new tree node to $T, \bar{T}$ and initialize $T(v)$ with the empty set and $\bar{T}(v)$ with all nodes from $V \setminus v$ as v is isolated in the beginning. This operation runs in time $\Theta(n)$. The running time is dominated by initializing the target tree of insertion.

Merging Clusters

For a *merge* operation $(C_1, C_2) \to C_3$ the edge probabilities for all intra-cluster edges of C_1 and C_2 are modified as these edges are now intra-cluster edges.

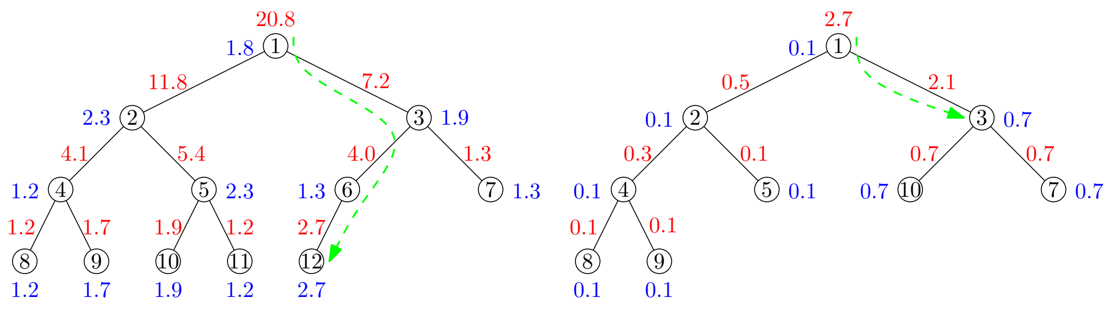
Therefore, an *update* operation for the source tree node of each $v \in C_1 \cup C_2$ becomes necessary. Additionally, the weight of all intra-cluster edges of C_3 has to be updated in the target trees because not only the weight of edges connecting C_1 and C_2 changed but also the intra-cluster edge probability $p_{\text{in}}(C_3)$ differs from the original values $p_{\text{in}}(C_1), p_{\text{in}}(C_2)$. The operation takes time $\mathcal{O}((n(C_1) + m(C_1) + n(C_2) + m(C_2)) \log n)$.

Splitting a Cluster

For the *split* operation $C_1 \to (C_2, C_3)$, the edge probabilities for all intra-cluster edges of C_1 are modified and some former intra-cluster edges inside C_1 are now inter-cluster edges of C_2 and C_3. Therefore, an *update* operation for the source tree nodes and the whole target trees of each $v \in C$ becomes necessary. Further points mentioned for *merge* apply accordingly causing a *split* operation to take time $\mathcal{O}((n(C_1) + m(C_1)) \log n)$.

Example: Inserting an Edge

Before we come to a short evaluation of this approach we would like to give another example illustrating the *insert* operation for edges. The figures of this example have been provided by Robert Görke and Christian Staudt. For this example we accept the respective weights of the tree nodes to be given. The graph to which the source tree belongs and a more detailed form of this example can be found on page 18 of [18].



(a) Selection of source node in the *source tree* $\bar{T}$ (b) Selection of target node *target tree* $\bar{T}(12)$

Figure 1.5.: Example of source and target trees: The weight of tree nodes is located next to the specific tree node (blue). Accumulated weights are placed next to the edge located above the respective tree (red). The dashed (green) arrow depicts the way which the algorithm takes for the given offset values.

As we already saw how the binary tree's *select* operation works (see example in Section 1.5.1) we do not describe the selection of edge $\{3, 12\}$ in detail but only point out the major steps:

- We start at the root of the *source tree for insertion* $\bar{T}$ (Figure 1.5(a)).
- We choose offset 12.7 and arrive at node 12. This is the source node of the edge to be created. Note that the target tree of node 12 has the same weight as the tree node 12.
- Inside the *target tree for insertion* $\bar{T}(12)$ (Figure 1.5(b)) we choose the offset to be 1.3 and the *select* procedure returns node 3. Note that edge tree nodes of the target tree have either weight 0.1 or 0.7 which can be identified as p_{out} and $p_{\text{in}}(c(12))$, respectively.

- We have finally found our next edge to insert: $\{3, 12\}$

As a next step – which is not depicted in the figures – we now have to remove node 3 from $\bar{T}(12)$ and node 12 from $\bar{T}(3)$ (as the non-edge $\{3, 12\}$ no longer exists) and we update the weight of nodes 12 and 3 and the accumulated weight all of their ancestors, i. e. , nodes 6, 3 and 1, in $\bar{T}$.

In the original work it was proved that, if we choose values in the range of $[0, \text{weight}(\text{root}))$ uniformly at random, the probability of selecting a certain tree node is proportional to its weight ω. We did not repeat the proof in this section but show an alternative representation of the above target tree in Figure 1.6 which immediately makes this claim evident. This interval representation can be easily generated from the above target tree by running a pre-order traversal, that is, arriving at a certain tree node we consider its weight ω first, descend into the left and finally into the right child tree.

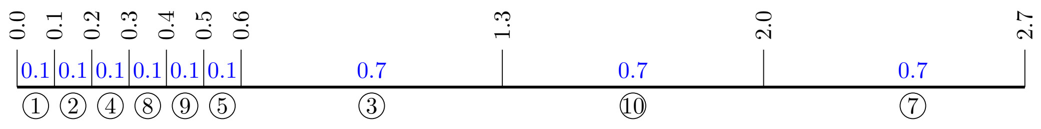


Figure 1.6.: Subdivisioning of the interval $[0, 2.7)$ corresponding to the target tree in the Figure 1.5(b)

1.5.3. Summary

We have seen that the source and target tree approach is very elegant as one only needs to know the binary selection tree in order to understand the functional principle. All graph operations run efficiently in $\mathcal{O}(\log n)$.

The major drawback of this approach is that it always needs memory of size $\Theta(n^2)$ because all possible $\binom{V}{2}$ edges are permanently stored either in the target trees for insertion or deletion. What may disturb us further is the logarithmic factor in the running time of the cluster operations. In the next chapter of this work we describe a different approach which redeems us of the space inefficiency.

1.5.4. Requirements

We summarize the requirements the new generator has to meet. From a functional point of view, the data structure must offer the following operations:

Edge Operations :

- select a pair of unconnected nodes $\{u, v\} \in \bar{E}$ and insert the edge $e = \{u, v\}$ into E
- select and delete an existing edge $e \in E$

Node Operations :

- create a node v, select a cluster C for v and insert v into C
- inserting a node may be combined with connecting v to other nodes
- select and delete an existing node $v \in V$ also removing all incident edges

Cluster Operations :

- select and split a cluster $C, |C| \geq 2$
- select two clusters and merge them

We want to create random networks. To this end, the following requirements constrain the dynamics with respect to selection probabilities.

- The decision which graph entity (node / edge) to insert or to delete is probabilistic.
- Starting with graph G, it must be possible to generate any adjacent graph $G' \sim G$.
- The probability for edge $e = \{u, v\}$ to be inserted must be proportional to $p(u, v)$ if $e \notin E_t$ or 0 else.
- The probability for edge $e = \{u, v\}$ to be deleted must be proportional to $1 - p(u, v)$ if $e \in E_t$ or 0 else.
- The probability for nodes to be deleted must be uniform for all nodes.

The data structure should fulfill the following performance guarantees. Space efficiency is the fundamental objective of this work. Concerning time guarantees the generator should be at least as fast as the previous implementation.

- The data structure must have a size significantly smaller than $\mathcal{O}(n^2)$ (ideally: $\mathcal{O}(n + m)$ – *space efficiency*).
- Edge insertion and deletion must run in time $\mathcal{O}(\log n)$ (ideally: $\mathcal{O}(1)$ – *time efficiency*).
- Insertion and deletion of node v must run in time $\mathcal{O}(n)$ (ideally: $\mathcal{O}(\deg(v))$ – *time efficiency*).
- Merging clusters C_1, C_2 should take at most time $\mathcal{O}(n(C_1) + n(C_2) + m(C_1) + m(C_2) + k)$
- Splitting cluster C should take at most time $\mathcal{O}(n(C) + m(C) + k)$

2. Cluster Tree Approach

In the previous section it became obvious that the source and target tree approach is limited by its space consumption: Even though we generally generate sparse graphs where the number of edges m is in $\mathcal{O}(n)$ the source and target trees permanently store the complete graph in memory, thus consuming memory $\Theta(n^2)$.

One of the most important observations is that the binary selection tree is a data structure which is more powerful than it needs to be: We will see, that the generation process can be reorganized such that it can profit from the regular structure of inter-cluster and intra-cluster edge probabilities. We will adhere to the generation process as depicted in Figure 1.1 – only the subprocedures will completely be reworked.

As this chapter is quite extensive it starts with a short overview of the following sections:

- Starting with the basics we explain first how nodes and edges are indexed in Section 2.1.
- Afterwards, a general purpose data structure called *Fisher-Yates shuffle* is described and enhanced with additional functionality (Section 2.2). This data structure will be used for edge dynamics in the subsequent section.
- The following sections are organized by type of operation. The first one, Section 2.3, deals with *edge dynamics*. As in the original approach edge dynamics are the most critical part of the generator. Once implemented, the other operations are quite easy to implement.
- Section 2.4 tackles node insertion and deletion, the so-called *node dynamics*.
- Section 2.5 finally describes cluster operations. in .

2.1. Node and Edge Indices

A well-designed scheme for indexing nodes and edges is the prerequisite of efficient graph operations. We propose a scheme which is widely used for storing graphs. What comes in handy is the fact that this scheme can be perfectly combined with the data structures presented later in this chapter.

2.1.1. Node Indices

In the following, we assume that we have n nodes in the graph under consideration the indices of which are running from 0 to $n-1$. Inside of a ground truth cluster this index is called the *local index* of a node. When a node is deleted from the a cluster, another node will be relabeled to fill the gap, thus keeping the index space *continuous*, that is, for each integer i in the range of 0 to $n-1$ we find an existing node the index of which is i. For the sake of simplicity we do not differentiate between the nodes and their indices whenever this is possible.

For our clustered dynamic graph, each node also possesses a *global index* which unambiguously identifies the node during the whole generation process – we do not reuse indices of deleted nodes! Nodes are inserted and deleted over time and thus the global index space is not continuous.

There exist two functions[1] which permit converting one index into the other:

- globalID(v) returns the global index of v.
- localID(v) returns the local index of v within its ground truth cluster

2.1.2. Triangular Edge Indexing Scheme

Since we only consider simple undirected graphs, only the left lower triangular submatrix of the adjacency matrix is relevant for numbering edges. Edges are indexed by traversing the submatrix line by line. The numbering scheme is illustrated by Figure 2.1. We see that the diagonal axis is not indexed as the graph does not contain self-loops.

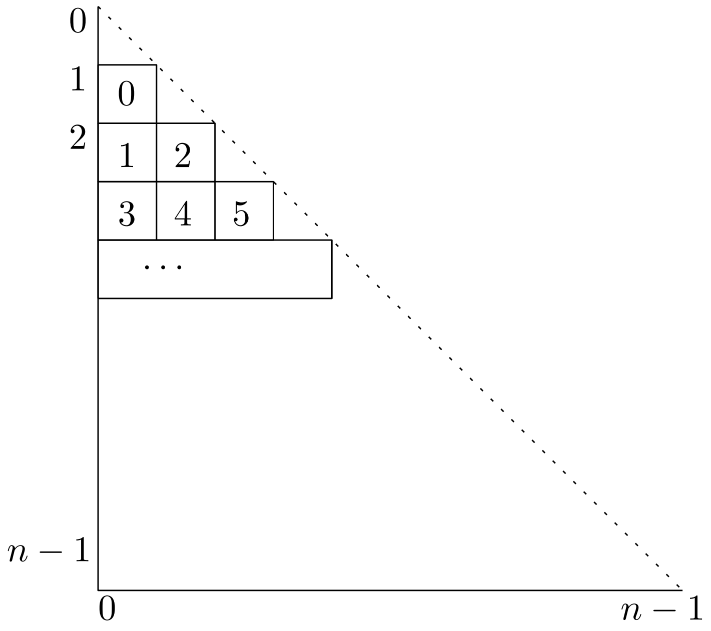


Figure 2.1.: Triangular indexing scheme

For given nodes $u, v \in V, u < v$ the index $e(u,v)$ of $e = \{u,v\}$ can be determined as follows:

$$\begin{aligned} e(u,v) &= \sum_{k=0}^{u-1} k + v \qquad (2.1.1) \\ &= \frac{1}{2}(u-1)u + v \end{aligned}$$

[1] For theoretical analysis we only consider the local index of a node within its ground truth cluster. Of course, when implementing the generator we need to keep track of the node's position within a potential reference cluster and of its position in the pseudo cluster.

Vice versa, given the edge index $e(u, v)$ the corresponding node indices u and v can be found as follows:

$$u = 1 + \left\lfloor -\frac{1}{2} + \sqrt{\frac{1}{4} + 2 \cdot e(u, v)} \right\rfloor$$
$$v = e(u, v) - \frac{1}{2}u(u-1)$$

2.1.3. Global and Local Indices of Edges

The edge index of edge $e = \{u, v\}$, which unambiguously identifies the edge over the whole generation process, can be obtained using the triangular indexing scheme from above applied to the global indices of u and v, respectively. Assuming that globalID$(u) >$ globalID(v) we get:

$$\text{globalID}(e) = \frac{1}{2}\,\text{globalID}(u) \cdot (\text{globalID}(u) - 1) + \text{globalID}(v).$$

2.2. Fisher-Yates Shuffle

Introducing the *Fisher-Yates shuffle* we prepare for the edge dynamics. The shuffle is a versatile tool and so we dedicated a separate section to it even though the shuffle will only be used when we consider the edge dynamics of the process in the next section. It has first been introduced as a pen-and-paper method for creating randomly permuted subsets from a given set of elements. We begin by describing the original data structure called the *standard Fisher-Yates shuffle* in Section 2.2.1 which dates back to the year 1948. Even though this version of the shuffle can be used without modifications in software but it always needs a worst-case amount of memory which is proportional to the number of the elements to choose from.

We must find a way to make the memory consumption proportional to the amount of actually selected elements and as, a second step, the data structure needs to be adapted in order to be able to also *delete* elements instead of always only adding them. The first point has already been tackled by Batagelj and Brandes who called the modified shuffle a *virtual* Fisher-Yates shuffle which is described in Section 2.2.2. The second point has been solved in this work. As the new shuffle offers more "dynamic" behavior we called it the *dynamic virtual* Fisher-Yates shuffle and show how the new method can be implemented in Sections 2.2.3 and 2.2.4.

2.2.1. Standard Fisher-Yates Shuffle

The (standard) Fisher-Yates shuffle [9] was originally designed as an efficient data structure for creating random permutations of k elements out of n. Note that n denotes the number of elements in the shuffle in the course of this section.

Fisher and Yates designed their algorithm to be carried out with pen and paper and thus all n elements were stored in "memory", that is, written down on a piece of paper.

In our imagination, we think of the shuffle as an array s. The array is divided into *slots* which contain *elements*. The content of slot x can be addressed by means of the bracket operator: $s[x]$. In our context elements are unsigned integers in the range of 0 to $n-1$. In the beginning, the elements shall be sorted in ascending order so that element x can be found in slot x: $s[x] = x$. In this case we say that x is *in place*.

The state of the shuffle is fully determined by the array s and the *border index* i. This index marks where we can find the *selected* and the *unselected elements* of the shuffle. The elements which have already been selected are stored at indices smaller than i (*left part*) and all element which may still be selected are stored at indices greater or equal to i ((*right part*). Consequently, $i = 0$ in the beginning because no element has been selected, yet. For $i < n$ the element $s[i]$ is called *border element* and the *border* is the separation of slot $s[i]$ and $s[i-1]$. The border element is undefined when all elements have already been selected, i. e., $i = n$. For the boundary cases where $i = 0$ and $i = n$ the border is located just before the first and just behind the last slot, respectively. Note that i is always identical to the number of selected elements. The shuffle offers only one operation: *select*. This operation draws an element from the set of unselected elements and adds it to the set of selected elements. In the rest of this section we will assume that we are able to select an element, that is, $i < n$.

The algorithm for deciding which element will be selected next can easily be understood: In each step we randomly select an index r in the range of $i, \ldots, n-1$ and swap this element with the border element. Finally, we move the border one slot to the right. Algorithm 4 implements the functionality in pseudo code. Each loop iteration corresponds to a *select* operation.

Algorithm 4: Standard Fisher-Yates Shuffle

```
Input  : $n$: maximum number of elements to select from, $k \leq n$: number of
         elements to be selected
Output : $s[0..k-1]$: selected elements
$s = [0, 1, \ldots, n-1]$: Array of Integer
for $i \leftarrow 0$ to $k-1$ do
    // Draw integer uniformly at random from the set $\{i, i+1, \ldots, n-1\}$
    r ← randomInt $(i, n-1)$
    swap $(s[i], s[r])$
```

Example

Before continuing with the first improvement concerning space consumption we give a small example (Figure 2.2) Starting with a shuffle of size $n = 5$ a permutation of $k = 3$ integers shall be produced. We only describe the first step as the subsequent two selections are analogous. During the first iteration, the border index is 0 and we choose $j = 1$ uniformly at random from the set $\{0, 1, 2, 3, 4\}$. The elements in slot 0 and 1 are swapped causing element 0 to be located in slot 1 and vice versa. Finally the border moves one slot to the right and the first selection is complete. After another two steps the resulting three-element permutation is $(1, 4, 2)$.

2.2.2. Virtual Fisher-Yates Shuffle

There are situation were we cannot afford to store all n elements in memory. This is the case when n is large in comparison to the number k of elements which we actually will draw. This situation can be tackled by means of a modification proposed by Batagelj and Brandes which they called *virtual Fisher-Yates shuffle* [2]. An important observation is that, generally speaking, most elements stay *in place* if $k \ll n$. So we only need to store the “exceptions to the rule”.

We make the following two changes in comparison to the standard Fisher-Yates shuffle:

- The selected elements – those that are located in the left part of the standard Fisher-Yates shuffle – are stored in memory, e. g. , in an unbounded array s of size i. This allows for fast access to the generated permutation.

- The unselected elements only exist in memory as two integers i and n which define the left and right border of the range to draw from and as a hash table which contains the so-called *replace pointers*. These pointers mark the indices where an element is not *in place*. Informally speaking, replace$(x) = y$ can be expressed as “Element y is now stored in slot x.” or “Element y replaces element x.”.

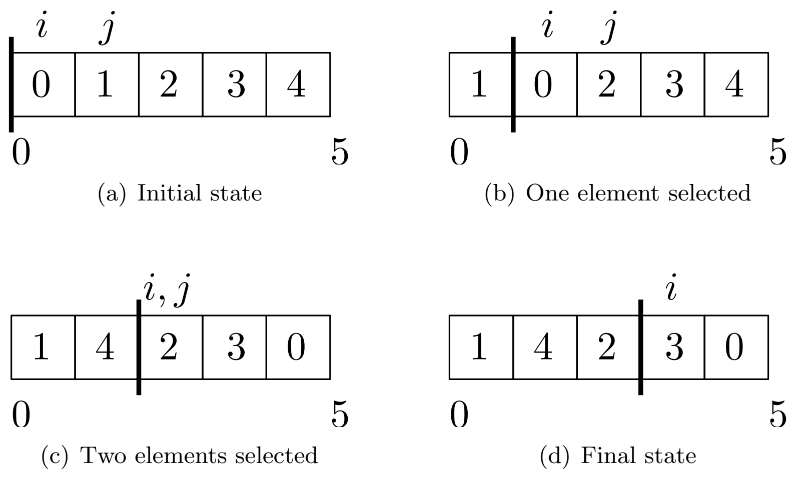


Figure 2.2.: Example of standard Fisher-Yates shuffle: i is the *border index*, the bold line is the *border*, and j is the index which has been drawn for the next *select* operation

If we query the hash table for elements k that are still *in place* the hash table returns replace(k) $=\perp$. When we now select a random index j, we first take a look at the replace table for index j. If a replacement exists, that is, replace(j) $\neq\perp$, we select replace(j) or otherwise we use j. There is one point to bear in mind: The border element may be different from i because the border index may already have been selected in a previous iteration. We therefore also have to take a look into the replace table for index i. In any case we have to update the replace table for index j: replace(j) := $s[i]$. Algorithm 5 summarizes the procedure. It is an adapted version of ALG. 3 in [2].

Algorithm 5: Virtual Fisher-Yates Shuffle

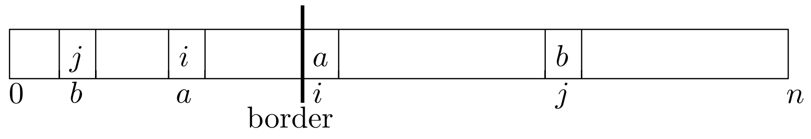
```
Input  : $n$: number of elements to select from, $k \leq n$: number of elements to be selected
Output : $s[0..k-1]$: the selected elements
$s$: Unbounded Array of Integer
replace: Hash Table of Integer $\to$ Integer
for $i \leftarrow 0$ to $k-1$ do
    // Draw integer uniformly at random from the set $\{i, i+1, \ldots, n-1\}$
    $j \leftarrow$ randomInt $(i, n-1)$
    if replace($j$) $\neq\perp$ then
        $s[i] \leftarrow$ replace($j$)
    else
        $s[i] \leftarrow j$
    if replace($i$) $\neq\perp$ then
        replace($j$) $\leftarrow$ replace($i$)
        replace($i$) $=\perp$
    else
        replace($j$) $\leftarrow i$
```

Setting replace(i) to $\perp$ is an optimization: As we only add elements over time and never put one back into the set of selectable elements, the border index i is rising strictly monotonic and so we can never draw index i again. As a consequence, the replacement of i will never be used again – we can delete it and save memory.

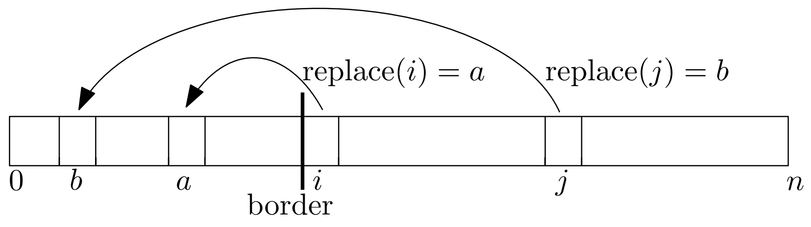
The general case where neither i nor j are *in place*, that is, replace(j) $\neq\perp\neq$ replace(i), is illustrated in Figure 2.3. The figure contains two representations of the same cut-out of a shuffle:

1. *Explicit Notation*: We depict the contents of each slot. In Figure 2.3(a) a is stored at index i.
2. *Pointer Notation*: We can use unidirectional replace pointers. The arrow points at the replacement of the index at the other end (see Figure 2.3(b)). The pointer starting at index j denotes that element b is now stored at index j and is equivalent to the entry replace(j) $= b$ in the hash table.

From now on, we will only use the pointer notation which makes illustrations more concise. Additionally, pointers are closer to the actual implementation.



(a) *Explicit Notation* using slots and elements



(b) *Pointer Notation* using replace pointers

Figure 2.3.: Virtual Fisher-Yates shuffle with two different types of representations

2.2.3. Dynamic Virtual Fisher-Yates Shuffle

One aspect which plays an important role for the original Fisher-Yates shuffle is the order of the elements (e. g. 12345 and 21435 are distinct permutations of the same elements). In our scenario, we do not care about the order in which the elements are drawn. Speaking in stochastic terms, we are satisfied with *combinations* instead of permutations.

Consequently, we do not have to keep the original order of the selected elements, but we can sort them at our convenience. The same applies for the right hand side of the shuffle: Due to the fact that each element on this side is equally probable to be selected, we can rearrange all elements which are not in place. For example, in Figure 2.3 we could swap a and b by updating the replace pointers accordingly without spoiling the result.

In contrast to the standard and virtual Fisher-Yates shuffle, we make the replace pointer bidirectional in order to allow element deletions. A pointer having endpoints x and y is abbreviated with $x \leftrightarrow y$ and its semantics is "x is stored in slot y and y is stored in slot x".

As before, the shuffle is fully determined by the maximum element count n, the border index i and the replace pointers. We define the *size* of a shuffle as the number of replace pointers. For the further discussion we need to define the equivalence relation for dynamic virtual Fisher-Yates shuffles:

Definition 2. *For two dynamic virtual Fisher-Yates shuffles $\mathcal{F}_1, \mathcal{F}_2$ let S_1, S_2 be the sets of selected elements and U_1, U_2 be the sets of unselected elements of $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively. We say that $\mathcal{F}_1$ is* equivalent *to $\mathcal{F}_2$ if both contain the same selected and unselected elements:*

$$\mathcal{F}_1 \textit{ equivalent to } \mathcal{F}_2 :\Leftrightarrow S_1 = S_2 \wedge U_1 = U_2.$$

Note that unlike before we do not need any data structure explicitly storing the selected elements. If we want to get the set of selected elements we simply iterate over all indices $x < i$. Unless we find that replace(x) $\neq \perp$ we add x to our result set and otherwise replace(x). For all further consideration, we suppose, that element access through the replace pointers runs in expected constant time and that (pseudo-)random numbers can be generated in constant time.

The fact that the order of elements within either part of the shuffle does not matter lets us rearrange the shuffle such that it fulfills the following property:

Lemma 1 (Replace pointers span the border)**.** *Any Fisher-Yates shuffle can be rearranged so that each replace pointer spans the border.*

We say that a replace pointer $a \leftrightarrow b$ spans the border, if a and b are on different sides of the border: $a < i \wedge b \geq i$ or $a \geq i \wedge b > i$.

Proof. Without loss of generality, suppose we have a shuffle as depicted in Figure 2.4 comprising the replace pointer $a \leftrightarrow b$. Translating the pointer into prose we obtain that a is stored in slot b and b is stored in slot a.

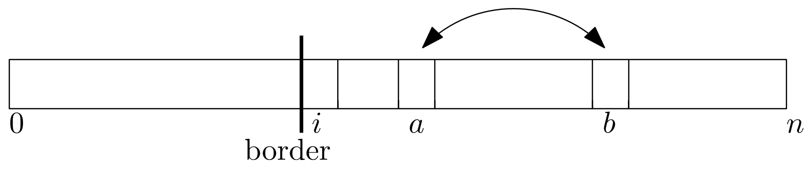


Figure 2.4.: Initial state of dynamic Fisher-Yates shuffle

We immediately see that – in the sense of Definition 2 – this shuffle is equivalent to Figure 2.5 where the contents of slot a and b have been swapped and a and b are *in place*.

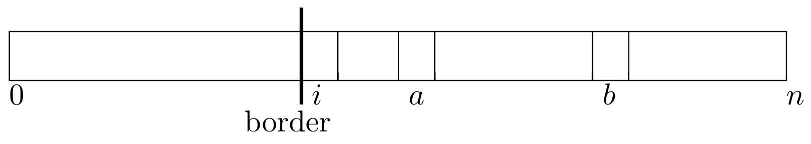


Figure 2.5.: Size-reduced dynamic Fisher-Yates shuffle

The same argumentation holds for the case when a and b are in the left part of the shuffle. □

Corollary 1. *Each dynamic virtual Fisher-Yates shuffle can be rearranged such that there are at most i replace pointers.*

Proof. As we can rearrange the shuffle so that no replace pointer spans the border and as by construction each index is pointed at by at most one replace pointer we find that the i indices to the left of the border can be endpoints of at most i replace pointers. □

This corollary is indeed quite useful as it states that we are able to transform an arbitrary shuffle such that its size is $\mathcal{O}$(#selected elements). The question is how we can keep the shuffle space efficient while the shuffle is modified by *select* and *delete* operations.

2.2.4. Operations on the Dynamic Virtual Fisher-Yates Shuffle

Four distinct cases exist for both operations and each case is depicted Figures 2.6 and 2.7 with an initial and a final state, respectively. They are sorted by the number of initial replace pointers of the relevant indices i, j (for *select*) and i', j (for *delete*). As before index j is the index which has been drawn uniformly at random from all feasible indices. Implementations of both procedures are straightforward and can be found in Algorithms 8 and 9 both of which are located in the appendix.

The time and size constraints for the *select* and *delete* operations are stated in Lemma 2.

Lemma 2 (*delete* and *select* for dynamic virtual Fisher-Yates shuffle)**.** *Selection and deletion can be implemented in expected constant time keeping the size of the shuffle linear in the number of selected elements.*

Proof.

Running Time: For either operation, each possible case needs only at most a constant number of pointer and arithmetic operations each of which runs in (expected) constant time. Thus, *delete* and *select* run in overall expected constant time.

Memory: This part of the proof is based on Corollary 1. We argue that we perform *select* and *delete* in a way that prevents replace pointers non spanning the border from being created or retained.

Clearly, this is true for the empty shuffle as it contains no replace pointers. During an operation the only replace pointers which are modified are the replace pointers for i, j (for *select*) and i', j (for *delete*). The case-by-case analysis in Figures 2.6 and 2.7 shows that each replace pointer existing after the operation and ending at one of these indices spans the border. □

Example

Analogous to Section 2.2.1 we give an example of how the space efficient Fisher-Yates shuffle works for the following parameters which are identical to the previous example (Figure 2.8): We have a maximum number $n = 5$ of selectable elements and we will choose $k = 3$ elements. The resulting three-element *combination* is $\{1, 2, 4\}$.

There are some aspects to mention about this example:

- As a reminder we argue that the only information actually held in memory are the values i, n and the replace pointers which are depicted as double-ended arrows. The memory consumption is proportional to the number of grey slots and we can see that each grey slot to the left of the border has a corresponding grey slot on the right side.

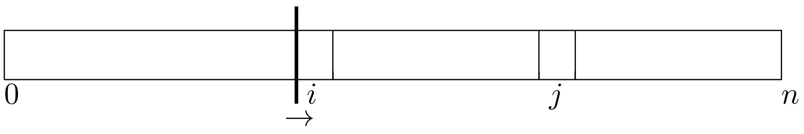
- The behavior of the shuffle differs from the standard Fisher-Yates shuffle for the first time when selecting the second element (Figure 2.8(b) $\rightarrow$ 2.8(c)): Index and element 1 are now on the left side of the border and so we may save one replace pointer by rearranging 1 and 4 in order to move element 1 *in place.*

- Similarly, if we consider the third step (Figure 2.8(c) $\rightarrow$ 2.8(d)) we see that no new replace pointer is required as element 2 stays *in place.*

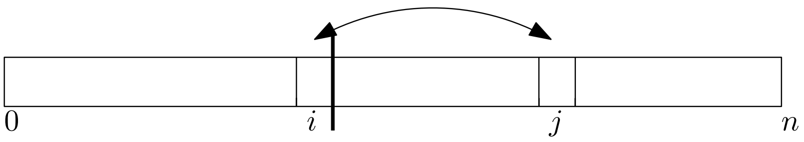
2.2.5. Summary

In this section we learnt about the Fisher-Yates shuffle and two modifications thereof. The Fisher-Yates shuffle is a data structure allowing for fast generation of permutations / combinations of n elements which are the integers $0, \ldots, n-1$ in our scenario. This implies that every element is equally probable to be selected.

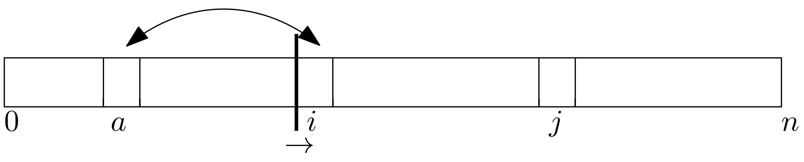
The original standard Fisher-Yates shuffle produces permutations and needs space $\Theta(n)$, where n is the maximum number of elements to select. The shuffle was designed to be



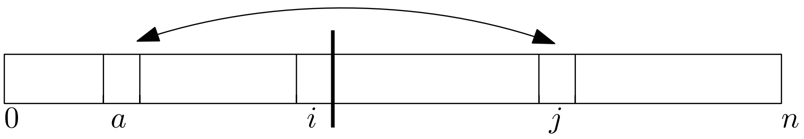
(a) Case 1: Initial state



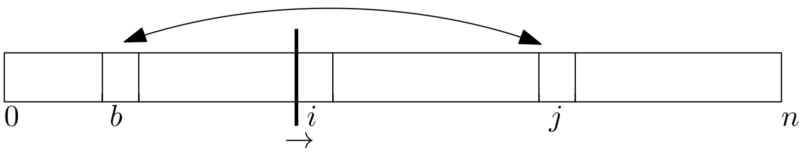
(b) Case 1: Final state



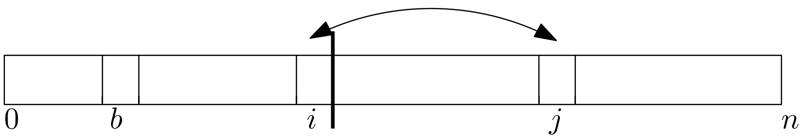
(c) Case 2: Initial state



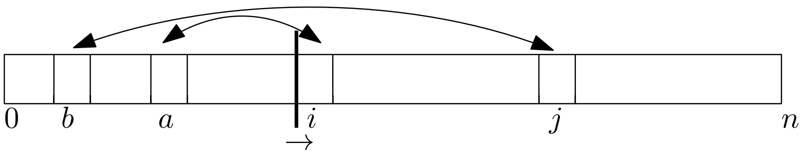
(d) Case 2: Final state



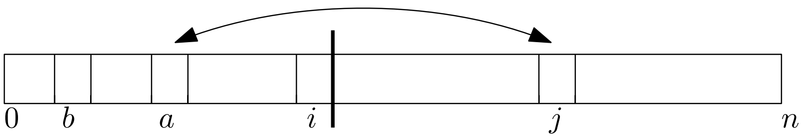
(e) Case 3: Initial state



(f) Case 3: Final state

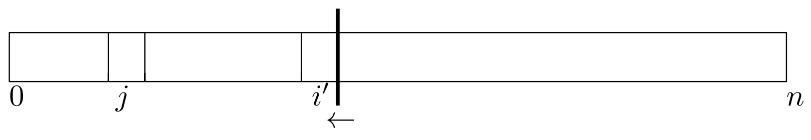


(g) Case 4: Initial state

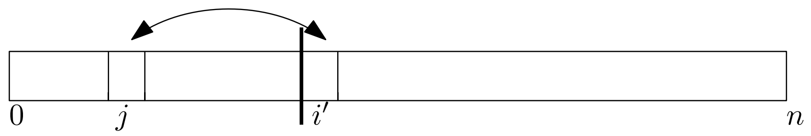


(h) Case 4: Final state

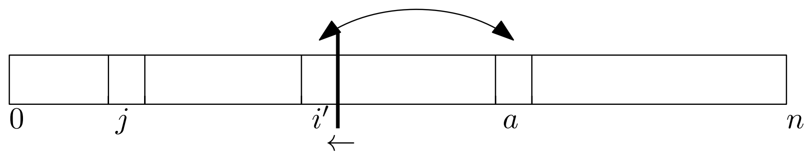
Figure 2.6.: Illustrative figures for *select*



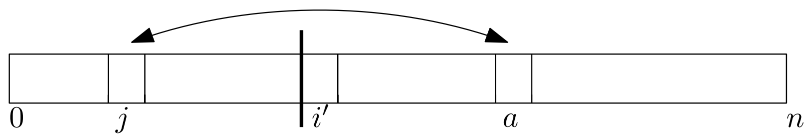
(a) Case 1: Initial state



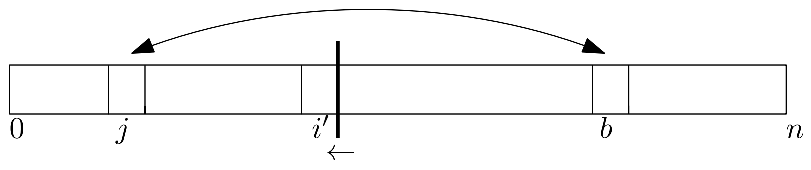
(b) Case 1: Final state



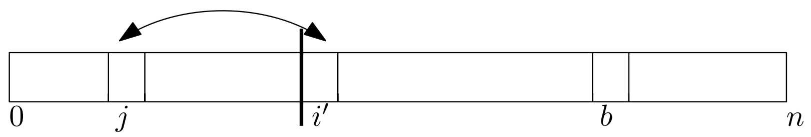
(c) Case 2: Initial state



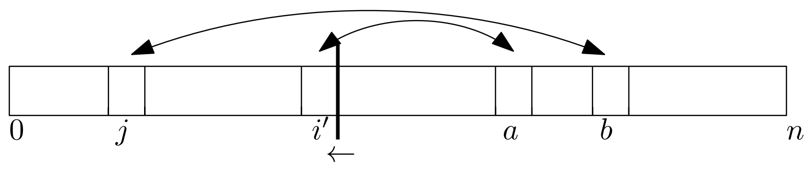
(d) Case 2: Final state



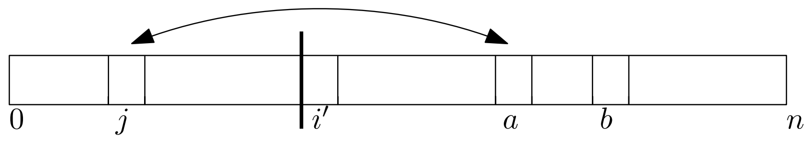
(e) Case 3: Initial state



(f) Case 3: Final state

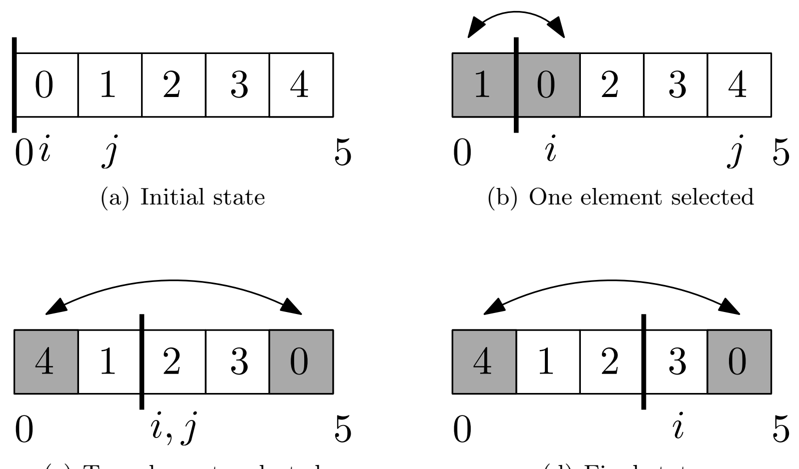


(g) Case 4: Initial state



(h) Case 4: Final state

Figure 2.7.: Illustrative figures for *delete*



(a) Initial state
(b) One element selected
(c) Two elements selected
(d) Final state

Figure 2.8.: Example of dynamic virtual Fisher-Yates shuffle: i is the *border index*, the bold line is the *border*, and j is the index which has been selected for the insertion. Grey slots are endpoints of *replace* pointers.

used for creating permutations of all n elements and so the constant memory consumption is acceptable.

The virtual Fisher-Yates shuffle is an improvement proposed by Batagelj and Brandes in [2] which was meant for adapting the shuffle to the scenario where we only select a small number $k \ll n$ out of all possible elements. This approach uses a hash table storing the indices at which we do not find the homonymous elements, the so-called *replace pointers.* The total memory consumption of the virtual Fisher-Yates shuffle is $\mathcal{O}(\#\text{selected elements})$. But still element deletion was not possible.

This limitation was removed by the modifications which we proposed: We basically use the same approach as Batagelj and Brandes but we organized the replace pointers such that they also support delete operations. Furthermore, through this modification we were able to represent the whole shuffle using only the hash table and two integers.

2.3. Edge Dynamics

After the preliminary work of the previous sections we are finally equipped with all the knowledge necessary to implement the actual operations. We start with edge operations as this kind of operation is most challenging to implement.

It has been mentioned before that the binary selection tree as described in Section 1.5.1 is a powerful data structure which can even be used in a scenario where the weight of each element is different. In our scenario, however, the quantity of distinct probabilities is limited to p_{out} and $p_{\text{in}}(C_i), C_i \in \mathcal{C}$ as can also be observed when taking a look at the target tree in the example for source and target trees in Section 1.5.2. This is where the generator can profit greatly from the dynamic virtual Fisher-Yates shuffle described in the previous section.

What may attenuate our hopes a little is the source tree of the same example: The weights seem to be completely irregular. In fact we cannot erase this irregularity but it can be limited to the relatively small set of clusters for which we will still use a binary selection tree.

The overall process of selecting a feasible edge for our operation is now divided into two steps:

1. We select a cluster, where the clusters are organized in two so-called *cluster trees*

$\Gamma_{\text{ins}}, \Gamma_{\text{del}}$. These trees are similar to the previous source trees for insertion and deletion. The name of our new approach stems from these trees.

2. We select our candidate within this cluster. This is done via a dynamic Fisher-Yates shuffle, where each cluster possesses a corresponding shuffle.

The structure of an ordinary cluster C is depicted in Figure 2.9.

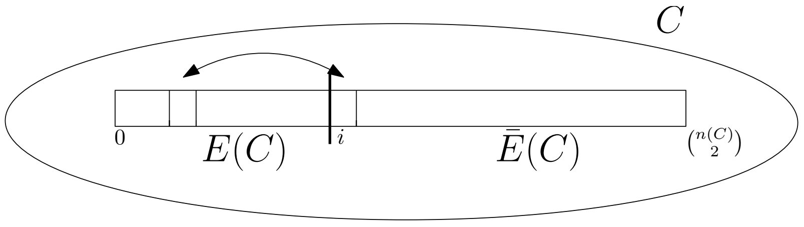


Figure 2.9.: Internal representation of edges in cluster C

The attentive reader has noticed an obvious problem arising from the first step of the strategy: Graphs also contain inter-cluster edges which cannot be assigned to a single cluster. For this reason, a new type of cluster is introduced in the following: the *pseudo cluster*.

2.3.1. The Pseudo Cluster

The pseudo cluster is defined as follows:

Definition 3 (Pseudo Cluster)**.** *For a given clustered Graph $G = (V, E)$ with clustering $\mathcal{C}$, the pseudo cluster C_0 contains all nodes of G, $C_0 = V$, and therewith $G(C_0) = G$.*

Clusters from the ground truth or reference clustering of G are called ordinary *clusters in order to stress the difference.*

The pseudo cluster possesses a data structure of size $\mathcal{O}(m_{\text{inter}})$ which separately stores the inter-cluster edges E_{inter}. In contrast to ordinary clusters it behaves differently from ordinary clusters when it comes to inserting or deleting edges:

- For *insert* operations the pseudo cluster chooses uniformly at random among all non-edges $\bar{E}$ which is the expected behavior.

- For *delete* operations, however, the choice is restricted to the set of inter-cluster edges E_{inter}.

This particularity will have to be taken into consideration when calculating the weight of C_0 for the cluster trees. The structure of the pseudo cluster is depicted in Figure 2.10. For the implementation of the additional data structure, an unbounded array is suitable because it allows for insertion, deletion and selection in amortized constant time as the order of inter-cluster edges does not matter because all inter-cluster edges are of uniform probability to be drawn.

Note that during *insert* we are now able to draw intra-cluster non-edges via C_0 and via the cluster containing the endpoints of the non-edge. The question may arise why we do not avoid this ambiguity and allow only for inter-cluster non-edges to be selected during *insert*. We will show what consequences this decision would have in the next but one paragraph, but first we need to take a look at the indexing scheme for C_0.

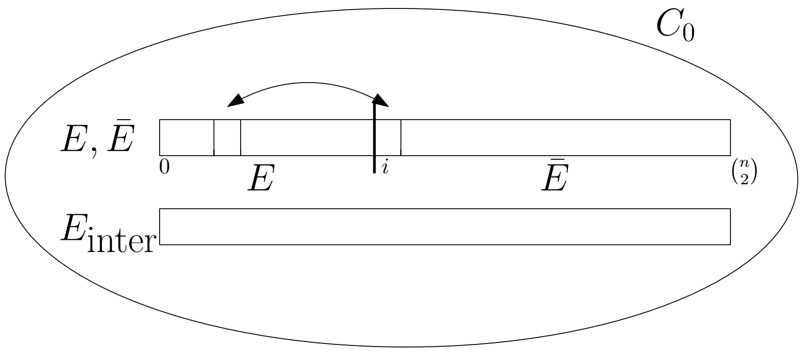


Figure 2.10.: Internal representation of edges in cluster C_0

Edge Indices within the Pseudo Cluster

At least for the *insert* operation, it is clear that any feasible edge can be selected within the pseudo cluster. That is why the triangular indexing scheme from Section 2.1 can be applied to C_0 seamlessly.

The definition of C_0 states that the *delete* operation works differently. This has no influence on the indexing scheme because the additional data structure which stores E_{inter} contains only global indices of the inter-cluster edges.

Alternative Interface for Edge Selection

Suppose now, we had defined C_0 to contain only the inter-cluster (non-)edges. The edge indices can be defined similarly to the ordinary clusters, using an adjacency matrix as depicted in Figure 2.11.

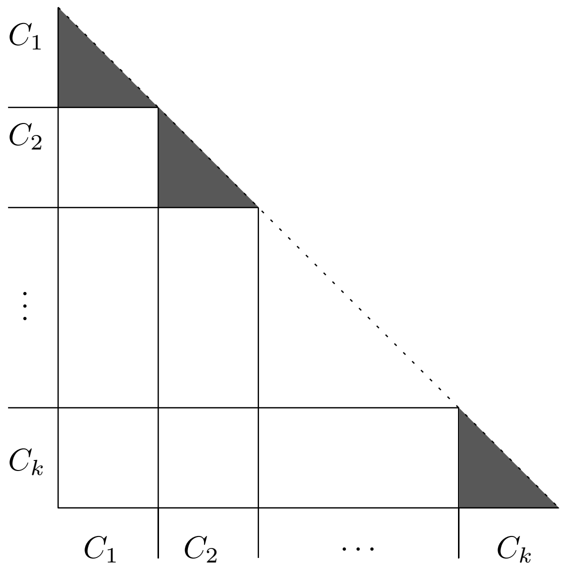


Figure 2.11.: Edge indices for alternative definition of C_0. As no intra-cluster edges are covered, the gray areas are not indexed.

The dark gray areas may not be selected as they represent the intra-cluster edges of the corresponding clusters. Therefore, the triangular indexing scheme could no longer be applied. Furthermore, node operations would be more tedious as the shape of the area which includes valid edge indices (white area) would be no more regular but change with every node operation. To our mind this problem would be more difficult to face than accepting an asymmetric interface.

2.3.2. Selection Process in Detail

Having cleared all open points, we now describe both steps in detail. Suppose we have decided for either an edge insertion or deletion. We are obliged to select a feasible candidate for this operation.

Step 1: Selecting the Cluster

For the former step, we will use binary selection trees as described in Section 1.5.1. The tree nodes are ordinary clusters and the pseudo cluster. One tree, Γ_{ins}, exists for *insert* and another one, Γ_{del}, for *delete* operations. For ordinary clusters $C_i, i = 1 \ldots k$, the weights $\omega_{\text{ins}}, \omega_{\text{del}}$ of a cluster in the insertion and deletion tree are defined as

$$\begin{aligned}\omega_{\text{ins}}(C_i) &= \bar{m}(C_i) \cdot (p_{\text{in}}(C_i) - p_{\text{out}})\\ \omega_{\text{del}}(C_i) &= m(C_i) \cdot (1 - p_{\text{in}}(C_i))\end{aligned}$$

We argue that $p_{\text{out}} \leq p_{\text{in}}(C_i)$ and therefore $p_{\text{in}}(C_i) - p_{\text{out}}$ is still a probability and the weight ω_{ins} for insertion is properly defined. The weight of C_0 is defined as:

$$\begin{aligned}\omega_{\text{ins}}(C_0) &= \bar{m}(C_0) \cdot p_{\text{out}} = \bar{m} \cdot p_{\text{out}}\\ \omega_{\text{del}}(C_0) &= m_{\text{inter}} \cdot (1 - p_{\text{out}})\end{aligned}$$

At this point it may not be clear why the weights are defined in exactly this way. Lemma 3 will (at least) clarify why these definitions lead to the desired results. As an intuition, we may say that we want to favor edges with relatively large edge probability during insertion and thus $\omega_{\text{ins}}(C)$ should be “proportional” to p_{in}. The reason for subtracting p_{out} in this term is that we may draw any intra-cluster non-edge also inside the pseudo cluster which has to be reflected by a reduced weight of the cluster which contains the given intra-cluster non-edge. Conversely, $\omega_{\text{del}}(C)$ should be “proportional” to $1 - p(u, v)$ as we prefer deleting edges with small edge probability. Furthermore, during *insert* clusters with a huge amount of intra-cluster non-edges should be favored over relatively dense clusters which leads to the fact that $\omega_{\text{ins}}(C)$ is proportional to $\bar{m}(C)$. Once again, the analogous argument applies for the deletion weight. The asymmetric behavior of the pseudo cluster becomes visible when comparing $\omega_{\text{ins}}(C_0)$ with ω_{del}: For a symmetric behavior, we would expect the edge count $m(C_0) = m$ to appear in the latter weight and not the inter-cluster edge count m_{inter}.

Step 2: Within a cluster

The next step is carried out on a Fisher-Yates shuffle belonging to the specific cluster. Most of the work for the actual edge selection has been done by adapting the Fisher-Yates shuffle in Section 2.2.3 in order to make it support the *delete* operation.

Each cluster $C_i, i = 0, \ldots, k$ possesses a virtual dynamic Fisher-Yates shuffle of its own. The elements in the shuffle of cluster C_i are all possible local edge indices $0, \ldots, \binom{n(C_i)}{2} - 1$.

Edge Insertion The Fisher-Yates shuffle offers the *select* method for selecting an unselected element which corresponds directly to selecting a non-edge for insertion. As we are able to draw the same intra-cluster non-edge inside the pseudo cluster and inside the cluster containing the endpoints we have to be careful to keep the Fisher-Yates shuffles of both clusters consistent.

Edge Deletion The situation is slightly different for edge deletions: Ordinary clusters behave as for the *insert* operation. Each edge may be chosen for deletion and the *delete* operation of the shuffle can directly be applied.

Even though all edges from E are contained in C_0 we assured that *only* the inter-cluster edges can be accessed by the *delete* operation on C_0. Therefore, *delete* does not select from the set of all edge indices in C_0 but only selects an entry in the additional data structure as illustrated in Figure 2.12.

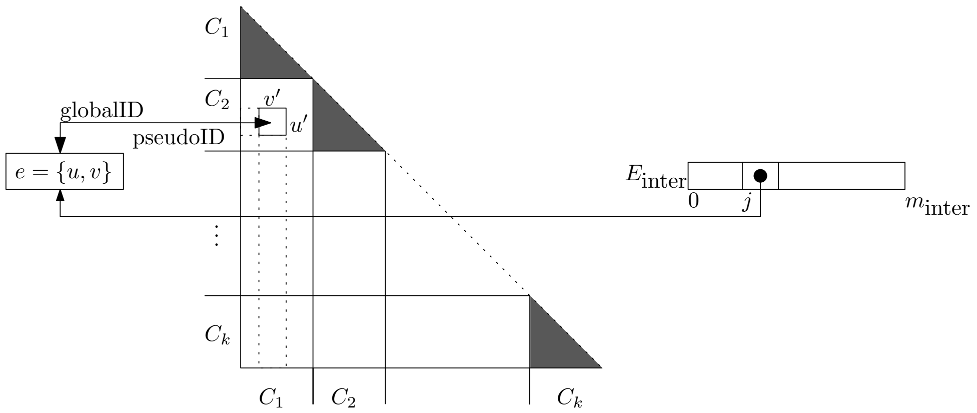


Figure 2.12.: Functional principle of *delete* in C_0: j is the integer drawn uniformly at random in the range of $0, \ldots, m_{\text{inter}} - 1$ and u', v' are the local and u, v are the global node indices of the endpoints of the selected edge. The *white* area consists of the inter-cluster edges and *dark gray* areas contains all intra-cluster edges.

Note, however, that the contents of the array labeled with E_{inter} are global edge indices this avoids relabeling when the local index of an edge changes. When edge $e = \{u, v\}$ is selected for deletion, its corresponding local indices within C_0 have to be calculated in order to remove the mapping from the Fisher-Yates shuffle.

Without loss of generality, the figure groups nodes by their corresponding clusters in order to depict the areas of inter-cluster (white) and intra-cluster (gray) adjacencies. Essentially, this indexing scheme is not the usual case as the indices of nodes within the same cluster may be arbitrarily distributed.

Proportional Probabilities

It remains to show that we truly obtain proportional probabilities, that is, the probability of a non-edge $\{u, v\}$ to be inserted is proportional to to its edge probability $p(u, v)$ and the probability of an edge to be deleted is proportional to $1 - p(u, v)$. The following lemmas may appear to be quite similar at first sight. However, due to asymmetric definitions of the weight in Γ_{ins} and Γ_{del}, the proofs differ slightly.

Lemma 3. *The* insert *operation selects an edge $e = \{u, v\}$ with probability proportional to $p(u, v)$.*

Proof. γ_i is the event that cluster C_i is selected and ε_e denotes the edge selection event for e.

General Observation: For the total weight of the insertion tree Γ_{ins} we find:

$$\begin{aligned}
\sum_{j=0}^{k} \omega_{\text{ins}}(C_j) &= \omega_{\text{ins}}(C_0) + \sum_{j=1}^{k} \omega_{\text{ins}}(C_j) \\
&= \bar{m} \cdot p_{\text{out}} + \sum_{j=1}^{k} \bar{m}(C_j) \cdot (p_{\text{in}}(C_j) - p_{\text{out}}) \\
&= (\bar{m}_{\text{inter}} + \underbrace{\bar{m}_{\text{intra}}}_{=\sum_{j=1}^{k} \bar{m}(C_j)}) \cdot p_{\text{out}} + \sum_{j=1}^{k} \bar{m}(C_j) \cdot p_{\text{in}}(C_j) - \sum_{j=1}^{k} \bar{m}(C_j) \cdot p_{\text{out}}) \\
&= \bar{m}_{\text{inter}} \cdot p_{\text{out}} + \sum_{j=1}^{k} \bar{m}(C_j) \cdot p_{\text{in}}(C_j) \\
&= \sum_{\{u,v\} \in \bar{E}_{\text{inter}}} p(u,v) + \sum_{\{u,v\} \in \bar{E}_{\text{intra}}} p(u,v) \\
&= \sum_{u \not\sim v} p(u,v)
\end{aligned}$$

The probability of an ordinary cluster C_i to be chosen in Γ_{ins} is proportional to the weight $\omega_{\text{ins}}(C_i)$:

$$\begin{aligned}
p(\gamma_i) &= \frac{\omega_{\text{ins}}(C_i)}{\sum_{j=0}^{k} \omega_{\text{ins}}(C_j)} \\
&= \frac{\omega_{\text{ins}}(C_i)}{\sum_{u \not\sim v} p(u,v)}
\end{aligned}$$

In step two, a Fisher-Yates shuffle is used and, once we are inside cluster C_i, the probability for each edge $e = \{u, v\}$ to be drawn is either 0 if $e \notin \bar{E}(C_i)$ or otherwise

$$p(\varepsilon_e|\gamma_i) = \frac{1}{\bar{m}(C_i)},$$

which is independent of the edge probability $p(u, v)$.

Intra-Cluster Edges: Remarkably, for an intra-cluster non-edge $e \in \bar{E}(C_i)$ there exist two possible sequences of events which entail a selection of e: $(\gamma_i, \varepsilon_e)$ and $(\gamma_0, \varepsilon_e)$. Therefore, the selection probability of e is:

$$\begin{aligned}
p(\varepsilon_e) &= p(\gamma_i) \cdot p(\varepsilon_e|\gamma_i) + p(\gamma_0) \cdot p(\varepsilon_e|\gamma_0) \\
&= \frac{\omega_{\text{ins}}(C_i)}{\sum_{u \not\sim v} p(u,v)} \cdot \frac{1}{\bar{m}(C_i)} + \frac{\omega_{\text{ins}}(C_0)}{\sum_{u \not\sim v} p(u,v)} \cdot \frac{1}{\bar{m}} \\
&= \frac{\bar{m}(C_i)(p_{\text{in}}(C_i) - p_{\text{out}})}{\sum_{u \not\sim v} p(u,v)} \cdot \frac{1}{\bar{m}(C_i)} + \frac{\bar{m} \cdot p_{\text{out}}}{\sum_{u \not\sim v} p(u,v)} \cdot \frac{1}{\bar{m}} \\
&= \frac{(p_{\text{in}}(C_i) - p_{\text{out}}) + p_{\text{out}}}{\sum_{u \not\sim v} p(u,v)} \\
&= \frac{p_{\text{in}}(C_i)}{\sum_{u \not\sim v} p(u,v)} \\
&= \frac{p(u,v)}{\sum_{u \not\sim v} p(u,v)}
\end{aligned}$$

Inter-Cluster Edges: For an inter-cluster non-edge $e \in \bar{E}_{\text{inter}}$ there exists only one possible sequence of events after which e is selected: $(\gamma_0, \varepsilon_e)$. The selection probability of e is:

$$\begin{aligned} p(\varepsilon_e) &= p(\gamma_0) \cdot p(\varepsilon_e|\gamma_0) \\ &= \frac{\omega_{\text{ins}}(C_0)}{\sum_{u \not\sim v} p(u,v)} \cdot \frac{1}{\bar{m}} \\ &= \frac{\bar{m} \cdot p_{\text{out}}}{\sum_{u \not\sim v} p(u,v)} \cdot \frac{1}{\bar{m}} \\ &= \frac{p_{\text{out}}}{\sum_{u \not\sim v} p(u,v)} \\ &= \frac{p(u,v)}{\sum_{u \not\sim v} p(u,v)} \end{aligned}$$

□

Lemma 4. *The* delete *operation selects an edge $e = \{u,v\}$ with probability proportional to $1 - p(u,v)$.*

Proof. γ_i is the event that cluster C_i is selected and ε_e denotes the edge selection event for e.

General Observation: For the total weight of the deletion tree Γ_{del} we obtain:

$$\begin{aligned} \sum_{j=0}^{k} \omega_{\text{del}}(C_j) &= \omega_{\text{del}}(C_0) + \sum_{j=1}^{k} \omega_{\text{del}}(C_j) \\ &= m_{\text{inter}} \cdot (1 - p_{\text{out}}) + \sum_{j=1}^{k} m(C_j) \cdot (1 - p_{\text{in}}(C_j)) \\ &= \sum_{\{u,v\} \in E_{\text{inter}}} (1 - p(u,v)) + \sum_{\{u,v\} \in E_{\text{intra}}} (1 - p(u,v)) \\ &= \sum_{u \sim v} (1 - p(u,v)) \end{aligned}$$

The probability of selecting cluster C_i in Γ_{del} is proportional to the weight $\omega_{\text{del}}(C_i)$:

$$p(\gamma_i) = \frac{\omega_{\text{del}}(C_i)}{\sum_{j=0}^{k} \omega_{\text{del}}(C_j)}$$

In step two, a Fisher-Yates shuffle is used and, once we are inside cluster C_i, the probability for each edge $e = \{u,v\}$ to be drawn is either 0 if $e \notin E(C_i)$ or otherwise

$$p(\varepsilon_e|\gamma_i) = \begin{cases} \frac{1}{m_{\text{inter}}} & i = 0 \\ \frac{1}{m(C_i)} & \text{otherwise} \end{cases}$$

Note that $p(\varepsilon_e|\gamma_i)$ distinguishes between ordinary and pseudo clusters.

Intra-Cluster Edges: An intra-cluster edge $e \in E(C_i)$ can only be selected if its containing cluster is chosen in Γ_{del} and if afterwards e is drawn from the shuffle which is abbreviated with the following sequence of events: $(\gamma_i, \varepsilon_e)$. Therefore, the selection probability of e is:

$$\begin{aligned} p(\varepsilon_e) &= p(\gamma_i) \cdot p(\varepsilon_e|\gamma_i) \\ &= \frac{\omega_{\text{del}}(C_i)}{\sum_{u\sim v}(1-p(u,v))} \cdot \frac{1}{m(C_i)} \\ &= \frac{m(C_i) \cdot (1-p_{\text{in}}(Ci))}{\sum_{u\sim v}(1-p(u,v))} \cdot \frac{1}{m(C_i)} \\ &= \frac{1-p_{\text{in}}(Ci)}{\sum_{u\sim v}(1-p(u,v))} \\ &= \frac{1-p(u,v)}{\sum_{u\sim v}(1-p(u,v))} \end{aligned}$$

Inter-Cluster Edges: For an inter-cluster edge e the following sequence of events is the sole one which results in a selection of e: $(\gamma_0, \varepsilon_e)$. We obtain:

$$\begin{aligned} p(\varepsilon_e) &= p(\gamma_0) \cdot p(\varepsilon_e|\gamma_0) \\ &= \frac{\omega_{\text{del}}(C_0)}{\sum_{u\sim v}(1-p(u,v))} \cdot \frac{1}{m_{\text{inter}}} \\ &= \frac{m_{\text{inter}} \cdot (1-p_{\text{out}})}{\sum_{u\sim v}(1-p(u,v))} \cdot \frac{1}{m_{\text{inter}}} \\ &= \frac{1-p_{\text{out}}}{\sum_{u\sim v}(1-p(u,v))} \\ &= \frac{1-p(u,v)}{\sum_{u\sim v}(1-p(u,v))} \end{aligned}$$

$\square$

2.3.3. Summarizing Example

The whole selection process is depicted in Figure 2.13. This figure is an example for a possible *insert* operation. The following list illustrates the sequence of decisions. For the sake of simplicity clusters are not distinguished from their corresponding tree node, for example $\omega_{\text{ins}}(C_i)$ is the weight of the tree node containing C_i and ℓ_i is the left child of that tree node.

- *Step 1: Cluster Tree*
  - $\Delta \in [0, \text{weight}(C_0))$ is drawn uniformly at random.
  - $\Delta < \text{weight}(C_x)$: descend into left subtree
  - $\Delta \geq \text{weight}(C_z) + \omega_{\text{ins}}(C_x)$: descend into right subtree with offset $\Delta' = \Delta - (\text{weight}(C_z) + \omega_{\text{ins}}(C_x))$.
  - $\text{weight}(\ell_y) \leq \Delta' < \text{weight}(r_y) + \omega(C_z)$: tree node containing C_y is selected
- *Step 2: Fisher-Yates shuffle*
  - $j \in \{i, \ldots, \binom{n(C_y)}{2} - 1\}$ is drawn uniformly at random
  - We find that $\text{replace}(i) \neq \perp$ and $\text{replace}(j) \neq \perp$ which is essentially the situation depicted in Figure 2.6(g)
  - Thus, the shuffle is reordered according to Figure 2.6(h).

The insertion of the new intra-cluster edge still has to be propagated to the pseudo cluster.

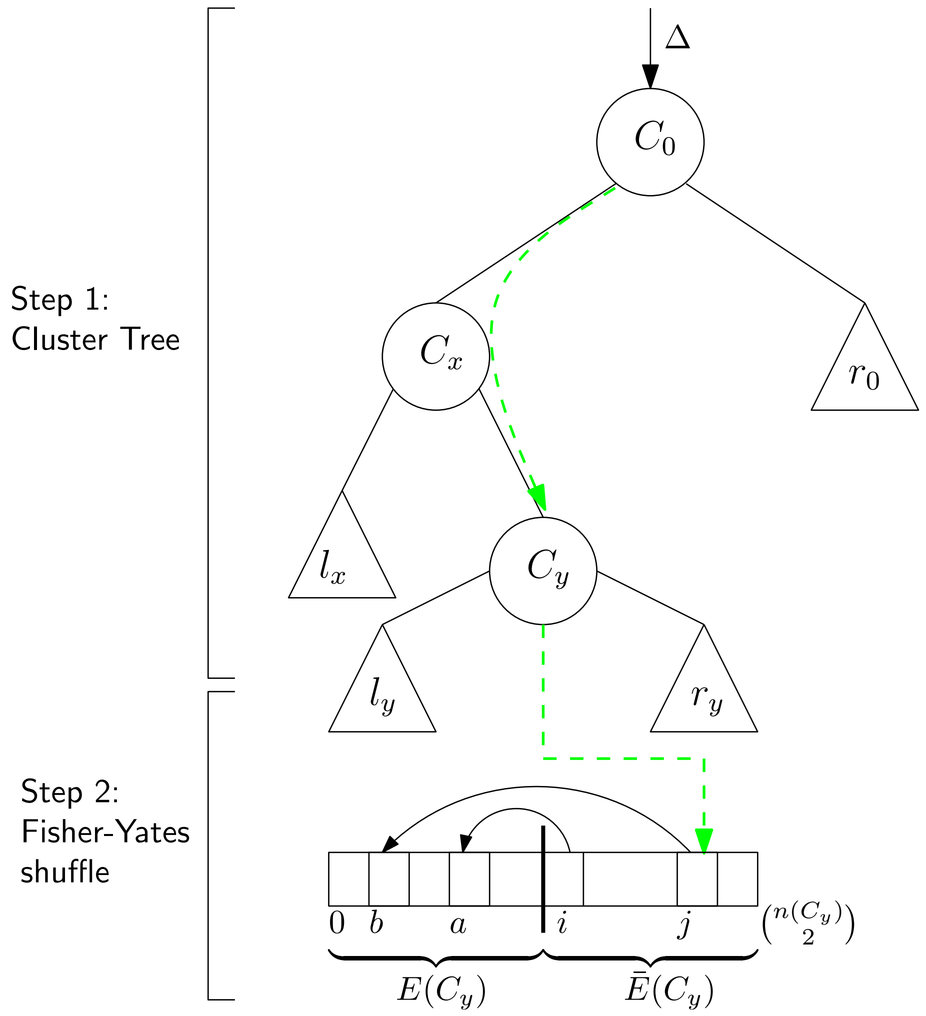


Figure 2.13.: Functional principle of the cluster tree. The path taken by the *insert* operation is drawn as dashed arrow (green).

2.4. Node Dynamics

In this section we will show how to implement the node operations. These operations partly build upon the edge operations described in the previous section.

2.4.1. Node Insertion

Generally, a new node is added to its containing cluster and to the pseudo cluster. A new local index is generated by appending a row to the adjacency matrix as is depicted in Figure 2.14.

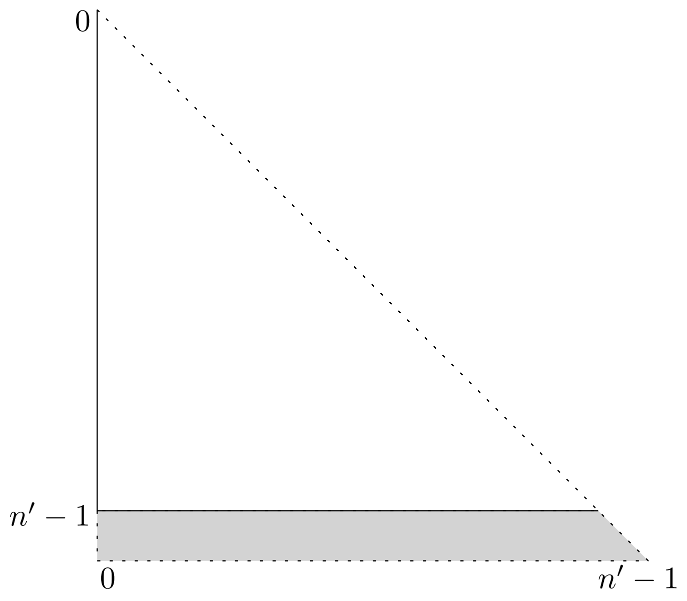


Figure 2.14.: Node insertion. n' is the new number of nodes: $n' = n + 1$.

The following steps are necessary for inserting a node into $G = (V, E)$:

1. Create new unique global node index v.
2. Select a cluster C which shall contain the new node v and add v to C: $C' = C + v$. This entails resizing the Fisher-Yates shuffle of C. In our model we would like to keep the initial distribution of cluster sizes so that the selection of a cluster has to consider this.
3. Adjust the Fisher-Yates shuffle of the pseudo cluster C_0.

We may assume that choosing a cluster for the new node takes time $\mathcal{O}(\log k)$ as we can organize the clusters in another binary tree where each cluster is weighted with its expected size. Furthermore, changing the node count of a cluster changes maximum edges count of C and of C_0 and therefore, the cluster trees have to be updated needs time $\mathcal{O}(2 \cdot \log k)$. All other operations run in constant time so that we get running time $\mathcal{O}(\log k)$ for a node insertion.

Creating Initial Adjacencies

After being added to the graph, a new node is isolated. This is a perfectly valid state of the graph. The fact that the degree of the new node differs (significantly) from the expected degree causes the generator to preferentially create adjacencies to the new node. However, we might not be willing to wait until the generator has steered the degree of the new node closer to the expected value and we decide to add edges by ourselves.

If we want to establish initial adjacencies of a new node v located in cluster C, we may iterate over all nodes within C and for node $u \in C - v$ the edge $\{u, v\}$ is created with probability $p_{\text{in}}(C)$. The same can be repeated for all nodes outside C, $u \in V \setminus C$, creating edges with probability p_{out}. This naive approach takes time $\mathcal{O}(n)$.

The so-called *geometric method* can be used to reduce the running time to $\mathcal{O}(\deg v)$. It has been introduced by Fan et al. [8] and Batagelj and Brandes [2] used it for efficiently generating graphs for Gilbert's model $\mathcal{G}(n, p)$.

In Section 1.4.3 the expected degree of a node in the $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$ model has been calculated and for the *insert* operation we obtain a running time of $\mathcal{O}(\deg_{\text{intra}}(C_i) + \deg_{\text{inter}} + \log k)$ which expands to

$$\mathcal{O}\left(p_{\text{in}}(C_i) \cdot (n(C_i) - 1) + \frac{p_{\text{out}}}{n} \sum_{i=1}^{k} n(C_i)(n - n(C_i)) + \log k\right)$$

2.4.2. Node Deletion

Node insertion and deletion are quite similar. There is one additional point which needs to be tackled: Deleting an arbitrary node u from cluster C_i results in a gap in the index space of nodes and edges. This gap is depicted as light gray areas in Figure 2.15. We can fill the gap by swapping the adjacencies of the last node v_f (dark gray areas) into the gap and relabel v_f to the local index of u.

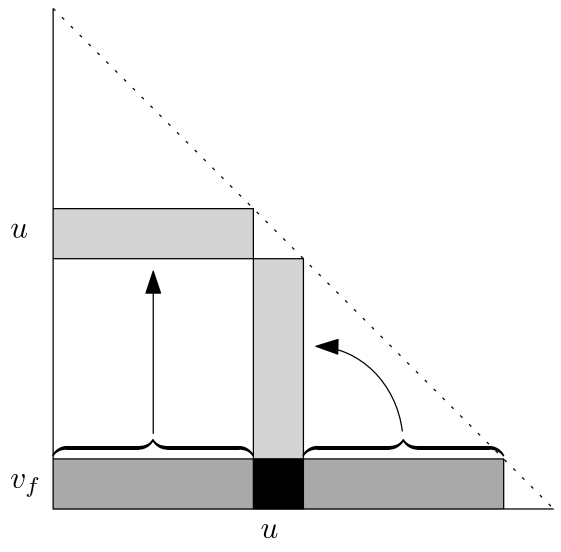


Figure 2.15.: Node deletion: *light gray*: invalidated adjacencies of u – *dark gray*: adjacencies to be copied from v_f – *black field*: edge $\{u, v_f\}$

In this case, too, we have to update the cluster trees which costs $\mathcal{O}(2 \cdot \log k)$. The inter-cluster edges of u are treated in the same way: We delete all incident edges in the pseudo cluster and swap the last node w_f of the pseudo cluster into the resulting gap. Finally, the extra data structure containing only E_{inter} must be updated. The running time for deleting node u is

$$\mathcal{O}(\underbrace{\deg_{\text{intra}} u + \deg_{\text{intra}} v_f}_{C_i} + \underbrace{\deg u + \deg w_f + \deg_{\text{inter}} u}_{C_0} + 2 \cdot \log k).$$

It is rather annoying that the running time also depends on the degree of the completely unrelated nodes v_f and w_f. At least, as we saw in Section 1.4.3 all nodes within a cluster have the same expected degree, so that we find the expected running time

$$\mathcal{O}(p_{\text{in}}(C_i) \cdot (n(C_i) - 1) + \deg w_f + \log k)$$

which is not fully satisfactory.

2.5. Cluster Dynamics

As a third and last type cluster operations are described in this section. Besides the description of the *split* and *merge* operation we need to take care of another point: When inserting a node we search for a cluster which shall contain the new node. The data structure supporting this decision has to be fast in order not to slow down this graph operation. Section 2.5.1 describes an efficient data structure.

The section is followed by the description of the *merge* (Section 2.5.2) and *split* (Section 2.5.3) operation. Additional detailed question that may arise during the reading of this section are mostly answered in Section 1.4.4 which describes the internal structure of the generator.

Without loss of generality we use the first cluster(s) of the ground truth for the operations. We work with these specific clusters rather than with unidentified ones as we could not demonstrate parts of the operations otherwise.

2.5.1. Expected Cluster Size

In contrast to its actual size, the expected size of a cluster remains constant over its lifetime. The expected size is set when a cluster is instantiated which takes place during the creation of the initial instance or a cluster operation. A sensible choice for storing the expected sizes would be a binary selection tree the tree nodes of which are the clusters which are weighted by their expected size. This choice allows for cluster selection, insertion and deletion to run in $\mathcal{O}(\log k)$.

2.5.2. Merging Clusters

We consider the operation $(C_1, C_2) \to C_{k+1}$. Afterwards we need to relabel the last two clusters in order to preserve a continuous range of indices. During the whole process, we keep copies of the original clusters C_1, C_2. Algorithm 6 states the different steps which need to be performed. Nevertheless, it is not a formal description but rather a high-level overview of what has do be done.

Algorithm 6: *merge* operation

```
// Create and insert new clusters
$C_{k+1} \leftarrow C_1 \cup C_2$
$\mathcal{C} \leftarrow \mathcal{C} - C_1 - C_2 + C_{k+1}$
$p_{\text{in}}(C_{k+1}) \leftarrow$ sample from current $p_{in}$ (see Section 1.4.4)
// Relabel nodes
foreach $v \in C_2$ do $\text{localID}(v) = n(C_1) + \text{localID}(v)$
create empty Fisher-Yates shuffle $\mathcal{S}$ for $C_{k+1}$
insert $E(C_1)$ into $\mathcal{S}$
insert $E(C_2)$ into $\mathcal{S}$
insert $E(C_1, C_2)$ into $\mathcal{S}$
// These edges are no longer inter-cluster edges
remove $E(C_1, C_2)$ from $E_{inter}$ in $C_0$
delete $C_1$ and $C_2$
calculate $\omega_{\text{ins}}(C_{k+1})$ and $\omega_{\text{del}}(C_{k+1})$
insert $C_{k+1}$ into $\Gamma_{\text{ins}}$
insert $C_{k+1}$ into $\Gamma_{\text{del}}$
// Preserve continuous indices by relabeling $C_k$ and $C_{k+1}$
$C_1 \leftarrow C_k$
$C_2 \leftarrow C_{k+1}$
```

Running Time

The node operations take

$$\mathcal{O}(n(C_1) + n(C_2)).$$

Filling the Fisher-Yates shuffle with all intra-cluster edges and with edges between C_1 and C_2 needs time

$$\mathcal{O}(m(C_1) + m(C_2) + m(C_1, C_2)) \text{ expected}.$$

Updating the former inter-cluster edges $E(C_1, C_2)$ in C_0 runs in

$$\mathcal{O}(m(C_1, C_2)) \text{ amortized}$$

Calculating and updating the weight of the tree nodes in $\Gamma_{\text{ins}}, \Gamma_{\text{del}}$ can be accomplished in

$$\mathcal{O}(m(C_1) + m(C_2) + m(C_1, C_2) + 2 \log k)$$

In sum, the *merge* operation runs in expected time

$$\mathcal{O}(n(C_1) + n(C_2) + m(C_1) + m(C_2) + \log k).$$

2.5.3. Splitting a Cluster

As expected, the *split* operation is similar to the *merge* operation. For the split operation $C_1 \to (C_{k+1}, C_{k+2})$ we initially create two new clusters C_{k+1} and C_{k+2} and decide on the desired values for $p_{\text{in}}(C_i)$ and how the nodes of C_1 are distributed onto C_{k+1} and C_{k+2}. Finally, C_{k+2} has to be relabeled in order to fill the gap in the list of indices. Algorithm 7 gives a rather high-level overview of the necessary operations. Note that we have to distribute the nodes of C_1 onto C_{k+1} and C_{k+2}. For this reason we have to sample the expected size for each of the new clusters.

Algorithm 7: *split* operation

```
// Create and insert new clusters
sample expected cluster sizes of $C_{k+1}, C_{k+2}$ (see Section 2.5.1)
$(C_{k+1}, C_{k+2}) \leftarrow$ partition nodes of $C_1$ according to expected sizes
$\mathcal{C} \leftarrow \mathcal{C} - C_1 + C_{k+1} + C_{k+2}$
$p_{\text{in}}(C_{k+1}) \leftarrow$ sample from current $p_{in}$ (see Section 1.4.4)
$p_{\text{in}}(C_{k+2}) \leftarrow$ sample from current $p_{in}$

re-assign $\text{localID}(v), v \in C_1$

create empty Fisher-Yates shuffles $\mathcal{S}_{k+1}, \mathcal{S}_{k+2}$ for $C_{k+1}, C_{k+2}$
insert $E_{k+1} = E(C) \cap (V(C_{k+1}) * V(C_{k+1}))$ into $\mathcal{S}_{k+1}$
insert $E_{k+2} = E(C) \cap (V(C_{k+2}) * V(C_{k+2}))$ into $\mathcal{S}_{k+2}$
insert $E(C_{k+1}, C_{k+2})$ into $E_{inter}$ in $C_0$

delete $C_1$
calculate $\omega_{\text{ins}}(C_{k+1}), \omega_{\text{ins}}(C_{k+2})$ and $\omega_{\text{del}}(C_{k+1}), \omega_{\text{del}}(C_{k+2})$
insert $C_{k+1}$ into the $\Gamma_{\text{ins}}$
insert $C_{k+2}$ into the $\Gamma_{\text{del}}$

// Preserve continuous indices by relabeling $C_{k+2}$
$C_1 \leftarrow C_{k+2}$
```

Running Time

We obtain the overall running time in a similar way as for the *merge* operation:

$$\mathcal{O}(n(C_1) + m(C_1) + \log k) \text{ expected}.$$

2.5.4. Summary of Cluster Operations

Cluster operations are the most extensive change we can make in the structure of the graph. They need a long sequence of single steps. Due to complexity we gave "road maps" for implementing cluster operations which also allowed us to derive the running time constraints which can be formulated as:

$$\mathcal{O}(n(C) + m(C) + \log k) \text{ expected}$$

where C is the largest one of the three clusters participating in the operation.

2.6. Review of the Cluster Tree Approach

We have now reached the end of this work's theoretical part. In the course of this chapter it became clear that the desired improvements in memory consumption can be reached at the cost of increased complexity. It can be expected that this complexity will lead to certain constant factors in the actual implementation.

Before proceeding from theory to practice we want to summarize this chapter. As in the previous implementation of the generator the key point for performance improvements are the edge operations. Therefore, this chapter started with this type of graph operation but not before we introduced two data structures which have not been used in the previous approach: the *pseudo cluster* and the *dynamic virtual Fisher-Yates shuffle*. After finishing with the edge operations we could go ahead with the node operations which are rather easy to design. Nonetheless, we had to be careful about necessary changes in the shuffle and the pseudo cluster when operating upon nodes. Up to now, the running time of deleting nodes is not fully satisfactory.

Finally, the cluster operations could use graph operations as a "black box". Of course, this type of operation has an enormous impact on the structure of the clustered graph. During cluster operations rather technical questions appeared such as the way the sizes of new clusters are sampled.

3. Implementation

We now come to the actual implementation of the generator described in theory above. The generator has been implemented in the widely used language *Java*. At first we make notes about different aspects of the implementation. In the second part we give results of the experimental evaluation. The generator is available at

`http://i11www.iti.uni-karlsruhe.de/en/projects/spp1307/dyngen`.

This site also contains information about how to download the generator.

3.1. Implementation Notes

This section consists of a set of different subtopics such as the specification of the command line arguments or the file format.

Please note: This generator is not meant for selling purposes and therefore does not fulfill industrial quality standards concerning stability against malformed inputs. Please read through this section carefully.

3.1.1. Command Line Parameters

Table 3.1 provides an overview of the command line options which are accepted by the generator. The formal grammar is given in Extended Backus-Naur form (EBNF). Parameters preceded with a dash are called *top-level parameters*. All other parameters are *configuration parameters*.

```
argument ::= "-h" | "-defaults" | "-g" configuration | "-f" infile
configuration ::= { keyval }
keyval ::= ikey "=" ival | dkey "=" dval | ilkey "=" ilist |
dlkey "=" dlist | skeyval | fkeyval | bkey "=" bval | verb | log
verb ::= "-v" | "-vv" | "-vvv"
log ::= "-l"
bkey ::= "binary"
bval ::= "true" | "false"
ikey ::= "n" | "t_max" | "k" | "eta
dkey ::= "p_out" | "p_mu" | "p_nu" | "p_omega" | "p_chi" |
"theta" | "beta" | "deg_in" | "deg_out""
ilkey ::= "deg_in_list" | "cl_sizes"
ilist ::= "[" ival { "," ival } "]"
dlkey ::= "p_in_list" | "deg_in_list"
dlist ::= "[" dval { "," dval } "]"
```

```
fkeyval = "dir=" directory | "output=" outfile
skeyval ::= "p_in_new=" p_in_new_method
```

The following restrictions apply for the variables which could not be defined in the grammar:

- `directory` is a valid directory name
- `infile` is the name of an existing file which contains a valid `configuration` on each line
- `outfile` is a valid filename
- `p_in_new_method` is the name of a supported method for calculating new p_{in} values during a cluster operation. For mor information see Section 1.4.4. Supported values are:
  - `GAUSSIAN` for Gaussian distribution and
  - `MEAN` for the arithmetic mean.

Table 3.1 gives an overview of the parameters with their respective domains.

The domains in the table only roughly restrict the range of possible values for each parameter as it is most often too cumbersome and not necessary to define the domain in an exact way. In some cases, the same information can be expressed with different (sets of) parameters. The following paragraphs will describe in how far several characteristics can be configured using different parameter sets.

Cluster Sizes

The expected size of the initial clusters can be set in the following ways:

- `k`,`n`:

  The expected size of each cluster is equal to $\frac{n}{k}$.

- `k`,`n` and `beta`:

  The clusters will have a *skewed* distribution where cluster $C_i, i = 1, \ldots, k$ has expected size $\sqrt[\beta]{\frac{n}{k}}$

- `n`, `cl_sizes`:

  The latter parameter defines the relative distribution of nodes onto the clusters (e. g. for `n=15` and `cl_sizes=[100,200]` the clusters will have expected sizes $\{5, 10\}$).

- `cl_sizes`:

  the elements of the list are directly interpreted as node counts by rounding the double values down. Note that therefore all elements have to be greater or equal to one.

Node Degree and Edge Probability

The intra-cluster edge probability p_{in} can be specified in several ways. Internally, a list of p_{in} is stored and will be passed to the generator. All other combinations of parameters are converted into this list. If the parameters `p_in_list` or `deg_in_list` appear it is necessary for those lists to have the length k where k can either be determined by `cl_sizes` or directly via the parameter `k`.

1. `p_in`:

   Each of the k clusters has the same intra-cluster edge probability p_{in}.

Top Level Parameters			
`-h`			prints help
`-v,-vv,-vvv`			activates verbose output with different granularity
`-l`			activates logging
`-g`			parameters for a single graph follow
`-f`			use each line of the following file a separate configuration
`-defaults`			print defaults
Configuration Parameters			
Key	**Equiv.**	**Domain**	**Description**
`n`	n_0	$\mathbb{N}$	initial node counter
`cl_sizes`		$\mathbb{R}_+^k$	relative cluster sizes
`k`	k	$\mathbb{N}$	number of clusters
`p_in`	$p_{\text{intra},0}$	$[0,1]$	intra-cluster edge probability
`deg_in`	$\deg_{\text{intra},0}$	$\mathbb{N}$	expected intra-cluster degree
`p_in_list`	$p_{\text{intra},0}$	$[0,1]^k$	p_{in} defined separately for each cluster
`deg_in_list`	$\deg_{\text{intra},0}$	$\mathbb{N}^k$	expected intra-cluster degree
`p_out`	p_{out}	$[0,1]$	inter-cluster edge probability
`deg_out`	$\deg_{\text{inter}}$	$\mathbb{N}$	expected inter-cluster degree
`t_max`	$t_{\max}$	$\mathbb{N}$	number of time steps
`p_chi`	p_χ	$[0,1]$	prop. of edge op. ($1-p_\chi$: node event)
`p_nu`	p_ν	$[0,1]$	prob. of *insert* (node op. has been selected previously, $1-p_\nu$: *delete*))
`p_omega`	p_ω	$[0,1]$	prop. of cluster op.
`p_mu`	p_μ	$[0,1]$	prob. of *merge* (cluster op. has been selected previously, $1-p_\mu$: *split*)
`theta`	θ	$[0,1]$	strictness for cluster completeness
`beta`	β	$\mathbb{R}$	biased selection coefficient
`eta`	η	$\mathbb{N}$	limit of batch size per time step
`p_in_new`		{GAUSSIAN,MEAN}	method for deriving new p_{in}
`dir`		String	output directory
`output`		String	output file
`binary`		`{true,false}`	toggles (binary) output file generation

Table 3.1.: Command line parameters of the generator

2. `p_in_list`:

   The list is of size k and it specifies the intra-cluster edge probability for each cluster.

3. `deg_in`:

   Each of the k clusters shall have the same expected intra-cluster degree $\mathbb{E}[\deg_{\text{intra}}(C_i)]$.

4. `deg_in_list`:

   The list is of size k and it specifies the expected intra-cluster degree $\mathbb{E}[\deg_{\text{intra}}(C_i)]$ for each single cluster.

What has been mentioned about intra-cluster edge probabilities analogously applies to the inter-cluster edge probability p_{out} and the corresponding inter-cluster node degree $\deg_{\text{inter}}$:

1. `p_out`:

   The inter-cluster edge probability p_{out}. It has to be smaller than any single p_{in} value.

2. `deg_out`:

   The intra-cluster node degree $\deg_{\text{inter}}$. The expected size of each cluster has to be set in this case.

The formula for converting $\deg_{\text{intra}}(C_i)$ to p_{in} and $\deg_{\text{inter}}$ to p_{out} can be found in Section 1.4.3. If the parsed values for `deg_in` or `deg_out` are too large for the given expected sizes, the maximum value will be chosen, that is, $p_{\text{in}} = 1.0$ or $p_{\text{out}} = 1.0$, respectively. Note that the latter case does not make sense and some care should be taken when choosing values for $\deg_{\text{intra}}$ and $\deg_{\text{inter}}$ because a change to p_{out} affects far more pairs of nodes than does the same change to any of the p_{in}.

Default values

The generator provides default settings for each of the parameters described in Table 3.1. The default values can be retrieved by calling the generator with parameter `-defaults`. The values have been chosen to create a small graph ($n = 60, t_{\max} = 100$) with two clusters of equal size ($|C_i| = 30$). In the expected case, the number of nodes stays constant over time ($p_\nu = 0.5$) and in half of the cases an edge operation is started ($p_\chi = 0.5$). Furthermore, the generator always generates an output file in default mode (`binary=true`).

Verbosity and Logging

The generator will produce no textual output in the default configuration which is handy if the generator is used within shell scripts. However, if you should be interested in more information about the current progress, the generator offers different levels of verbosity which can be activated with the switches `-v`, `-vv` and `-vvv`, where the different levels will cause the following information to be printed:

- `-v`
  - current argument set (useful for `-f` option),
  - start and end time of the generation process
- `-vv`
  - all information from level `-v`,
  - progress in steps of 5%
- `-vvv`
  - all information from level `-vv`,

– each operation (node, edge insertion and deletion, merge, split, check for completeness,...)

Moreover, the generator may also log information about the progress. Logging can be activated by means of the switch `-l` which will cause the generator to produce a log file labeled with the date and time when the generator was called. The generator will log all available information which corresponds to verbosity level `-vvv`.

3.1.2. Graph File Format

Currently, only the binary *GraphJ* format is supported. For running performance tests the generator can be configured to produce no output at all by setting `binary=false`.

GraphJ

We use the non-standardized, but practically well-proven binary file format *GraphJ* proposed by Staudt and Görke for their dynamic random graph generator [18].

Each *GraphJ* file consists of a list of events with their respective opcodes and parameters. The data types are primitive Java integers (`int` – 4 Bytes) for arguments and bytes (`byte` – 1 Byte) for *opcodes*. The file is divided into two main parts for the different types of operations: graph operations and cluster operations. Each part is subdivided into two lists: The first list contains the opcodes and the second list contains the parameters for the operations. The file structure is sketched in Table 3.2. The index g stands for *graph operations* and c for *cluster operations*. The type of an operation is represented by an opcode. Opcodes for graph operations are listed in Table 3.3 and opcodes for cluster operations are listed in Table 3.4.

Data Type	Count	Description
`int`	1	o_g: length of opcode array for node/edge operations
`int`	1	p_g: length of parameter array
`byte`	o_g	opcodes
`int`	p_g	parameters to the opcodes
`int`	1	o_c: length of opcode array for node/edge operations
`int`	1	p_c: length of parameter array
`byte`	o_c	opcodes
`int`	p_c	parameters to the opcodes

Table 3.2.: *GraphJ* file structure

Node indices and cluster indices run from 1, and can be derived from the order of events: *create node* for indexing the nodes and *set cluster* for indexing the clusters. This file format allows for a parser which is shorter than 20 lines of code as Listing 3.1 illustrates. The code has to be duplicated in order to parse the graph and cluster operations separately.

Operation	**Code**	**Arg. 0**	**Arg. 1**
create node u	1	C	C_{ref}
delete node u	2	u	-
create edge $\{u, v\}$	3	u	v
delete edge $\{u, v\}$	4	u	v
set cluster of u	5	u	C
set reference cluster of u	6	u	C_{ref}
next time step	7	-	-

Table 3.3.: GraphJ opcodes for graph operations

Operation	**Code**	**Arg. 0**	**Arg. 1**	**Arg. 2**
next step	0	–	–	–
merge $(C_1, C_2) \to C_3$	1	C_1	C_2	C_3
split $C_1 \to (C_2, C_3)$	2	C_1	C_2	C_3
merge done $(C_1, C_2) \to C_3$	3	C_1	C_2	C_3
split done $C_1 \to (C_2, C_3)$	4	C_1	C_2	C_3

Table 3.4.: GraphJ opcodes for cluster operations

```
File file = new File("filename");
FileInputStream fStream = new FileInputStream(file);
DataInputStream dataStream = new DataInputStream(fStream);

int opLength = dataStream.readInt();
int argsLength = dataStream.readInt();

ArrayList<Byte> opcodes = new ArrayList<Byte>(opLength);
ArrayList<Integer> args = new ArrayList<Integer>(argsLength);

for (int i = 0; i < opLength; ++i)
{
        opcodes.add(dataStream.readByte());
}

for (int i = 0; i < argsLength; ++i)
{
        args.add(dataStream.readInt());
}
```

.

Figure 3.1.: Code sample for parsing one part of a *GraphJ* file

3.1.3. The Difference between Theory and Practice

As usual when implementing theoretical results, there are some practical issues which needed not be considered in theory. The following lists some of these aspects:

- We represent undirected edges as a pair of directed edges. Concerning absolute value, both parts of an edge $\{u, v\}$ get the same global and local index. For the half-edge where the start node has a smaller index than the target node we use a negative sign in order to distinguish both parts.
- Each cluster possesses an own adjacency list which contains all of the intra-cluster edges and the outgoing half of each inter-cluster edge.
- As a node may be part of two ordinary clusters, namely one ground truth and one reference cluster, we need to keep track of its local index in both clusters. Therewith, we get another index, the *reference cluster index*.
- Furthermore each node is also stored in the pseudo cluster which allows for fast iteration over all nodes of the graph. This entails keeping track of another local index for each node, the *pseudo cluster index*.
- Efficiency is often obtained at the cost of increasing complexity. This rule of thumb also holds for the generator: For example, we needed to implement extra data structures for holding the clusters which are available for *split* and *merge* operations. Keeping referential integrity upright is rather challenging as the number of such lists increases.
- For efficiently measuring the completeness of a cluster operation we cached the inter-cluster edge count $m(C_1, C_2)$ of the participating pair of clusters as we could not efficiently read this measure from any data structure. Moreover, the values a and b also had to be adjusted if the node count of C_1 or C_2 changed.

3.2. Exemplary Calls

As a practical introduction to the generator we show a few exemplary calls. The generator can be started in several ways:

- *ldcrgen.sh* is a Bash script for Unixes
- *ldcrgen.bat* is a Windows batch script
- *java -jar ldcrgen.jar* is the platform independent but most verbose way

In the following we will only print the calls for the Bash script. Our first example will cause the generator to produce a help message and quit.

```
./ldcrgen.sh -h
```

After this functional test, we try to use the generator for productive work. A small, yet usable graph can be obtained using the default parameters.

```
./ldcrgen.sh -g
```

In the following, we assume that we want to generate a small graph with $n = 100$ nodes which are in the expected case distributed uniformly onto $k = 4$ clusters. We define the intra-cluster edge probabilities to be $p_{\text{in}} = (0.4, 0.5, 0.65, 0.7)$ and we want to have an inter-edge probability of $p_{\text{out}} = 0.08$. As we are not interested in the exact sequence of events, we leave away the switches `-v` and `-l`. The given scenario can be generated as follows:

```
./ldcrgen.sh -g n=100 k=4 p_in_list=[0.4,0.5,0.65,0.7] p_out=0.08
```

3.3. Compatibility with the `DCRGenerator`

We list here the differences between the syntax of the previous generator and this generator.

output directory:

- previously: `outDir`
- now: `dir`

output file:

- previously: `filename`
- now: `output`

cluster sizes

- previously: `D_s`
- now: `cl_sizes` – Furthermore, this parameter may now also be used to determine the total node count n.

method for guessing new p_{in} values

- previously: `enp=true|false`
- now: `p_in_new=GAUSSIAN|MEAN` – This is meant for future extensions by other distributions.

list of p_{in} values

- previously: `p_inList`
- now: `p_in_list`

Some parameters did not exist in the previous generator:

- `deg_in`: uniform intra-cluster node degree
- `deg_out`: inter-cluster node degree
- `deg_in_list`: individual intra-cluster node degree per cluster

The old generator also offered storing graphs in *GraphML* as this feature was used rather seldom it was left away in the new generator.

3.4. Evaluation

The complexity of the generator makes it difficult to prove the stated guarantees concerning running time and memory footprint by experimental evaluation. To our opinion, it is most important that the generator performs well in everyday work. In order to give the reader an idea of the expected time it takes to generate certain instances we made two experiments:

1. Given a fixed number of nodes, how does the program behave when increasing p_{in} which corresponds to an increase in the number of edges?
2. Given a fixed intra-cluster edge probability p_{in} how does the runtime behave when we increase the node count n?

For each experiment we fitted a regression curve but we do not give the exact parameters of the curve due to the reasons described above. Furthermore, apart from the default settings, we used the following parameters:

- cluster count k: 15
- number of time steps $t_{\max}$: 10000 (first experiment) and 1000 (second experiment)
- inter-cluster edge probability p_{out}: 0.01
- The resulting graph was not written to a file: `binary=false`.
- remaining parameters: default values as listed in Section 3.1.1

The experiments were run on a machine with 16GB memory and a Dual-Core AMD Opteron 2218 processor. We measured the time for 10 runs of each configuration and plot the arithmetic mean.

3.4.1. Fixed Node Count

The first experiment was performed with three different values for n: 1000, 5000 and 10000. The results are depicted in Figure 3.2.

We see that the increase in running time is roughly linear in the expected edge count. We cannot explain the time intercept which makes up most of the running time for the first case where we only have few nodes.

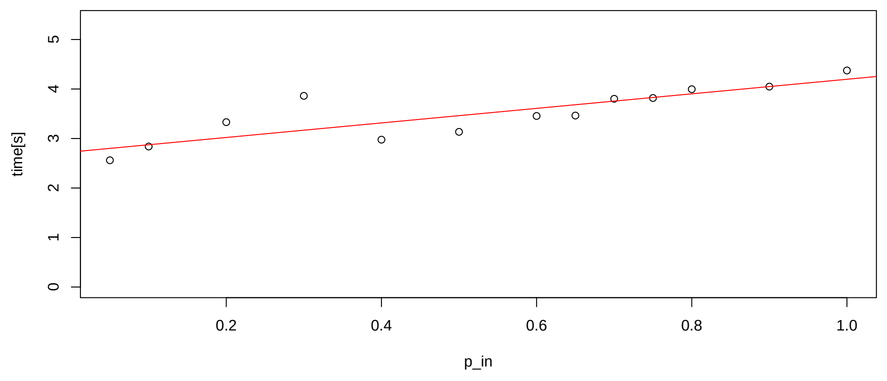
3.4.2. Fixed Intra-Cluster Edge Probability

The experiment was performed with three different values for p_{in}: 0.3, 0.6 and 0.9. The results are depicted in Figure 3.3.

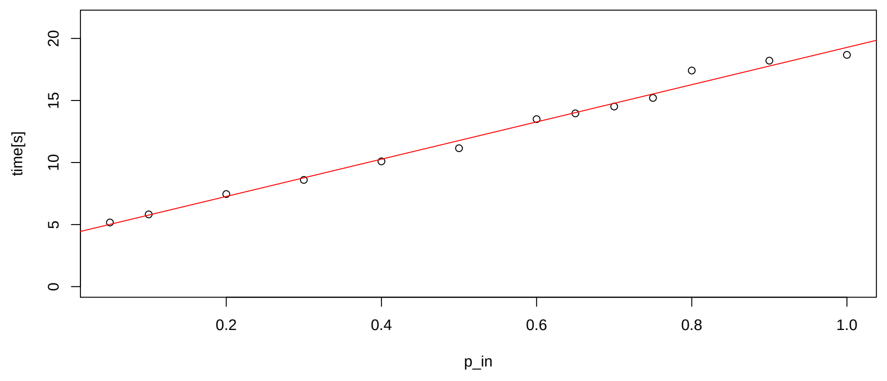
For a fixed intra-cluster edge probability we expect that the number of edge behaves like $\Theta(n^2)$. With some tolerance this expectation is met by the generator.

3.4.3. Summary

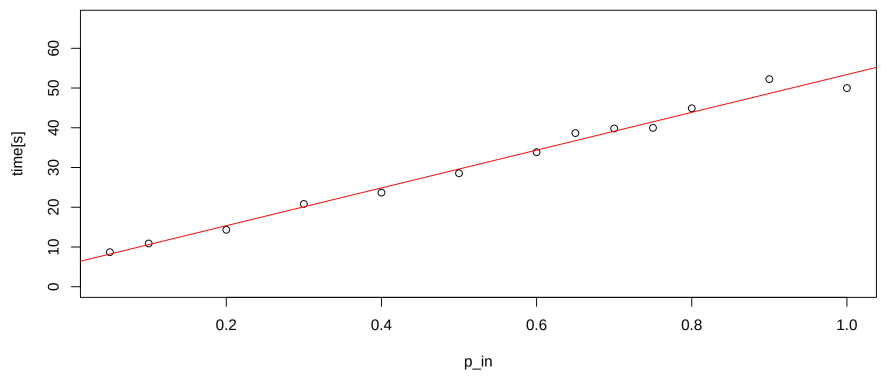
Even though we did not claim to proof any theoretical guarantee the experiments show that the generator generates test data of appropriate size within a reasonable time.



(a) $n = 1000$

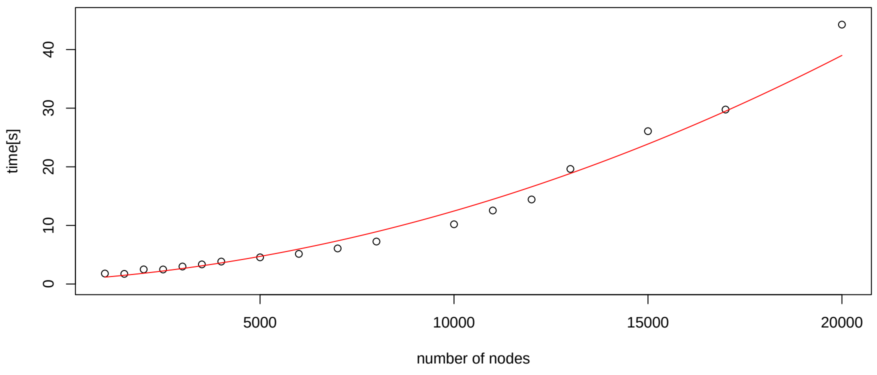


(b) $n = 5000$

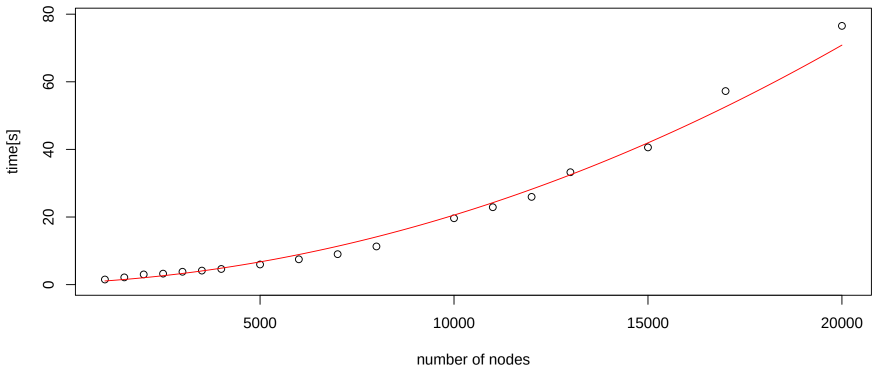


(c) $n = 10000$

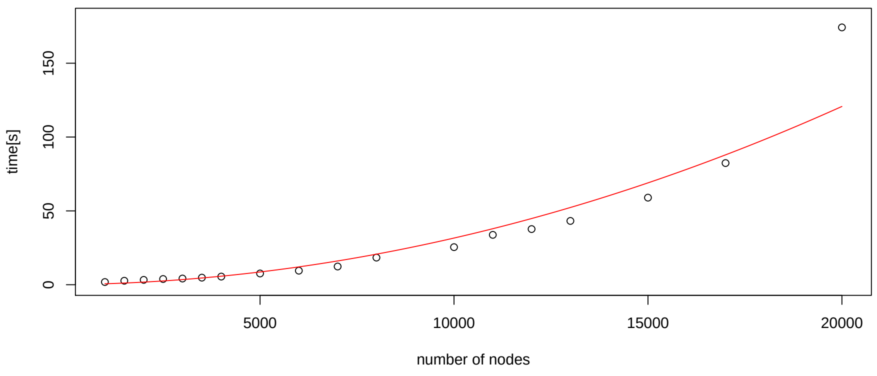
Figure 3.2.: First Experiment: Constant node count n and variable intra-cluster edge probability p_{in}. Values are averaged over 10 runs.



(a) $p_{\text{in}} = 0.3$



(b) $p_{\text{in}} = 0.6$



(c) $p_{\text{in}} = 0.9$

Figure 3.3.: Second Experiment: Constant intra-cluster edge probability p_{in} and variable node count n. Values are averaged over 10 runs.

4. Conclusion

Concluding this work, we would like to summarize what has been achieved so far and what can be subject to further research.

4.1. Summary

As a starting point, our objective was to generate test data for dynamic clustering algorithms. We found our work upon the widely used *intra-cluster density vs. inter-cluster sparsity paradigm* which is mirrored by the applied random graph model $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$. Robert Görke and Christian Staudt previously implemented a generator producing test data according to this model. Their generator is very elegant using only a few theoretical concepts and it behaves well in practice for small and medium-sized graphs. However, problems arise, when large and sparse test instances shall be created as the generator always takes space $\Theta(n^2)$. The problem can easily be illustrated comparing the behavior of the old generator with the new one. Say we generate a dynamic graph with 10,000 nodes distributed evenly onto two clusters, an intra-cluster edge probability of $p_{\text{in}} = 0.1$ resulting in an expected number of intra-cluster edges of $2 \cdot p_{\text{in}} \cdot \binom{5000}{2} \approx 2,400,000$ which is not too large a number. However, running both generators on a standard laptop with 2 GB of RAM assigned to the Java VM causes the old implementation to crash with an *OutOfMemoryError* whereas our new implementation needs approximately 40 seconds to generate the graph.

A fundamental observation is that the previous generator uses data structures which are "too" powerful for our purposes. We recognized that an edge operation can be composed of a weighted selection among the clusters and a uniform choice inside the cluster. For the first part we reused the *binary selection tree* which was used for the previous *source and target tree* approach. During this step we introduced the *pseudo cluster* which represents the inter-cluster edges in the tree. The weights of the tree nodes had to be chosen carefully in order to allow for proportional probabilities. Once, a cluster has been selected for an operation the *dynamic virtual Fisher-Yates shuffle* carries out the uniform choice of the affected (non-)edge. The shuffle allows for constant time element selection and deletion where each element is equally probable to be selected. The Fisher-Yates shuffle as it has originally been designed was only intended for inserting elements and needed space proportional to the maximum number of elements which can be drawn. The latter point being already solved by Batagelj and Brandes, we still needed to tackle the former one. By enhancing the concept of replace pointers we were able to represent the whole shuffle as a hash table and two integers. The design of the remaining operations was oriented at implementation of the edge operations.

Nonetheless, a lot of care had to be taken in order to assert referential integrity of the data structures in use which was one of the most challenging points during the implementation of the generator. Actually implementing the theoretical concepts helped to clarify ambiguous formulations within the thesis. Both, the implementation by Görke and Staudt and the new implementation are available at `http://i11www.iti.uni-karlsruhe.de/en/projects/spp1307/dyngen`.

We conclude this summary by giving an overview of the theoretical guarantees for running time and space consumption. We argue that all requirements as stated in Section 1.5.4 could be fulfilled.

4.1.1. Space Efficiency

The data structures can be divided into the following parts where the corresponding space consumption is given for each part:

- cluster trees $\Gamma_{\text{ins}}, \Gamma_{\text{del}}$: $\mathcal{O}(2 \cdot (k+1)) = \mathcal{O}(k)$
- cluster tree nodes for k ordinary clusters C_i: $\sum_{i=1}^{k} \mathcal{O}(n(C_i) + m(C_i))$
- cluster tree node for pseudo cluster C_0: $\mathcal{O}(n+m)$

As there are at most as many clusters as nodes this leads to a total memory consumption of $\mathcal{O}(n+m+k)$ which is equivalent to

$$\mathcal{O}(n+m)$$

in comparison to $\mathcal{O}(n^2)$ for the previous implementation.

4.1.2. Time Efficiency

Table 4.1 states the running time constraints for all cluster and graph operations. All running time constraints are expected values as we internally work with hash tables and unbounded arrays whereas the running times of the previous generator are deterministic. Even though it was not our foremost objective, we could also improve the running time of each single operation.

Operation	New Implementation	Previous Implementation
insert node v	$\mathcal{O}((1+\deg v) \cdot \log k)$	$\mathcal{O}(n)$
delete node v	$\mathcal{O}(\deg v + \deg v_f + \deg w_f + \log k)$	$\mathcal{O}(n)$
edge insertion	$\mathcal{O}(\log k)$	$\mathcal{O}(\log n)$
edge deletion	$\mathcal{O}(\log k)$	$\mathcal{O}(\log n)$
split $C \to (C_1, C_2)$	$\mathcal{O}(n(C) + m(C) + \log k)$	$\mathcal{O}((n(C) + m(C)) \cdot \log n)$
merge $(C_1, C_2) \to C$	$\mathcal{O}(n(C) + m(C) + \log k)$	$\mathcal{O}((n(C) + m(C)) \cdot \log n)$

Table 4.1.: Overview of running time guarantees: All bounds are *expected* values. v_f, w_f are the last nodes of $c(v)$ and the pseudo cluster, respectively.

4.2. Outlook and Open Questions

At the end of this section, we collect some open questions which arose during the work on this thesis. As the open points are spread all over the work the following list is a loose collection of ideas.

- During node deletion we choose a node uniformly at random from V because the random graph model puts no constraints upon this kind of dynamics. Perhaps, another approach could fit our intuition of "convergence" towards the desired ground truth in a better way: We assign a weight to each node $v \in V(C_i)$ which represents how much this node conforms to the ground truth, its "deviation" $d(v)$. As an idea the following metric could be used. It compares the number of inter-cluster and intra-cluster adjacencies of a node to the respective expected value:

$$d(v) = ||N_{\text{intra}}(v)| - \mathbb{E}[\deg_{\text{intra}}(v)]| + ||N_{\text{inter}}(v)| - \mathbb{E}[\deg_{\text{inter}}]|$$

- This generator has been implemented to generate graphs in the $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$ random graph model. Generators for dynamic test data in other models which have been described in Section 1.4.2 would be desirable. Maybe, some concepts of this generator could be used for simplifying the design of these generators.
- In Section 2.4.2 we saw that during a node deletion, we have to hard-copy the adjacencies of the last node in the cluster to their new positions in the Fisher-Yates shuffle. This implies that the running time of deleting an arbitrary node always depends on the degree of the last nodes of its cluster and the pseudo cluster. Perhaps a more sophisticated implementation of the shuffle or a completely different approach to the node operations might avoid this inconvenience.
- A similar problem occurs during cluster operations: Cluster operations currently build the new Fisher-Yates shuffle from scratch and insert the adjacencies afterwards. Perhaps, it might be possible to at least merge the shuffles. Solving this problem would not improve the asymptotic bounds but could lead to a significantly faster implementation of the cluster operations in practice.
- Finally, there is one theoretical aspect which could not be dealt with in the given time: Up to now, even though it seems to be intuitively obvious, the generator has not been formally proven to generate graphs according to $\mathcal{G}(n, p_{\text{in}}, p_{\text{out}})$. As previously mentioned a potential proof would be restricted to the implementation of edge operations as the random graph model states no constraints concerning node or cluster dynamics but presumes both to stay constant over time. An ansatz for the proof could be to model the generation process as a Markov chain.

Appendix

A. Algorithms

There is pseudo code which is not essential for understanding the functional principle of the described operations. For the sake of completeness we provide the procedures in this section. The algorithms are not optimized with respect to compactness. They are meant to mirror the different cases depicted in Figures 2.6 and 2.7. The *select* operation is implemented in Algorithm 8 and the *delete* operation in Algorithm 9.

The hash table represents each bidirectional replace pointer as two entries. If element j is still *in place* the hash table returns $\text{replace}(j) = \perp$. Setting $\text{replace}(j)$ to $\perp$ means removing the corresponding mapping from the hash table.

The state of the shuffle is represented in an object-oriented way: *select* and *delete* shall be methods of a class of which n, i and replace are attributes.

Algorithm 8: *select* operation

```
Attributes: $n$ : number of elements stored, $i$: border index, replace: Hash Table of
            Integer $\rightarrow$ Integer storing the replace pointers
Output    : $s$: the selected element
// Draw integer uniformly at random from the set $\{i, i+1, \ldots, n-1\}$
j $\leftarrow$ randomInt $(i, n-1)$
if $i \neq j$ then
    if replace($j$) $\neq \perp$ then
        $b \leftarrow$ replace($j$)
        if replace($i$) $\neq \perp$ then
            $a \leftarrow$ replace($i$)                         // Case 4
            replace($j$) $\leftarrow a$
            replace($a$) $\leftarrow j$
            replace($b$) $\leftarrow \perp$
            replace($i$) $\leftarrow \perp$
        else
            replace($i$) $\leftarrow j$                         // Case 3
            replace($j$) $\leftarrow i$
            replace($b$) $\leftarrow \perp$
        $s \leftarrow b$
    else
        $s \leftarrow j$
        if replace($i$) $\neq \perp$ then
            $a \leftarrow$ replace($i$)                         // Case 2
            replace($a$) $\leftarrow j$
            replace($j$) $\leftarrow a$
            replace($i$) $\leftarrow \perp$
        else
            replace($i$) $\leftarrow$ replace($j$)              // Case 1
            replace($j$) $\leftarrow$ replace($i$)
else if replace($i$) $\neq \perp$ then
    $s \leftarrow$ replace($i$)                      // Special Cases for $i = j$
    replace(replace($i$)) $\leftarrow \perp$
    replace($i$) $\leftarrow \perp$
else
    $s \leftarrow i$
$i \leftarrow i + 1$
```

Algorithm 9: *delete* operation

```
Attributes: $n$ : number of elements stored, $i$: border index, replace: Hash Table of
            Integer $\rightarrow$ Integer storing the replace pointer
Output    : $d$: the deleted element
$i' \leftarrow i - 1$
// Draw integer uniformly at random from the set $\{0, 1, \ldots, i'\}$
$j \leftarrow$ randomInt $(0, i')$
if $i' \neq j$ then
    if replace($j$) $\neq \perp$ then
        $b \leftarrow$ replace($j$)
        $d \leftarrow b$
        if replace($i'$) $\neq \perp$ then
            $a \leftarrow$ replace($i'$)
            replace($j$) $\leftarrow a$                         // Case 4
            replace($a$) $\leftarrow j$
            replace($b$) $\leftarrow \perp$
            replace($i'$) $\leftarrow \perp$
        else
            replace($i'$) $\leftarrow j$                        // Case 3
            replace($j$) $\leftrightarrow i$
            replace($b$) $\leftarrow \perp$
        replace($i'$) $\leftarrow \perp$
    else
        $d \leftarrow j$
        if replace($i'$) $\neq \perp$ then
            $a \leftarrow$ replace($i'$)
            replace($a$) $\leftarrow j$                         // Case 2
            replace($j$) $\leftarrow a$
            replace($i'$) $\leftarrow \perp$
        else
            replace($i'$) $\leftarrow j$                        // Case 1
            replace($j$) $\leftarrow i'$
else if replace($i'$) $\neq \perp$ then
    $d \leftarrow$ replace($i'$)                    // Special Cases for $i = j$
    replace(replace($i'$)) $\leftarrow \perp$
    replace($i'$) $\leftarrow \perp$
else
    $d \leftarrow i'$
$i \leftarrow i'$
```

Bibliography

[1] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[2] V. Batagelj and Ulrik Brandes. Efficient Generation of Large Random Networks. *Physical Review E*, 71(3):36113, 2005.

[3] Béla Bollobás, Oliver M. Riordan, Joel Spencer, and Gábor Tusnády. The Degree Sequence of a Scale-Free Random Graph Process. *Randoms Structures and Algorithms*, 18:279–290, 2001.

[4] Ulrik Brandes and Thomas Erlebach, editors. *Network Analysis: Methodological Foundations*, volume 3418 of *Lecture Notes in Computer Science*. Springer, February 2005.

[5] Ulrik Brandes, Marco Gaertler, and Dorothea Wagner. Experiments on graph clustering algorithms. In *Proceedings of the 11th European Symposium on Algorithms (ESA ´03) (LNCS 2832)*, ESA ´03, pages 568–579, 2003.

[6] Ulrik Brandes, Marco Gaertler, and Dorothea Wagner. Engineering Graph Clustering: Models and Experimental Evaluation. *ACM Journal of Experimental Algorithmics*, 12(1.1):1–26, 2007.

[7] P. Erdős and A. Rényi. On Random Graphs I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[8] C. T. Fan, Mervin E. Muller, and Ivan Rezucha. Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *Journal of the American Statistical Association*, 57:387–402, 1967.

[9] Ronald Aylmer Fisher and Frank Yates. *Statistical Tables for Biological, Agricultural and Medical Research*. Oliver and Boyd, London, 3rd edition, 1948.

[10] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3–5):75–174, 2009.

[11] OpenStreetMap Foundation. The OpenStreetMap Project. `http://www.openstreetmap.org/`.

[12] Marco Gaertler, Robert Görke, and Dorothea Wagner. Significance-driven graph clustering. In *Proceedings of the 3rd international conference on Algorithmic Aspects in Information and Management*, AAIM '07, pages 11–26, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] Horst Gilbert. Random Graphs. *The Annals of Mathematical Statistics*, 30:1141—-1144, 1959.

[14] Robert Görke. *An Algorithmic Walk from Static to Dynamic Graph Clustering*. PhD thesis, Karlsruhe Institute of Technology, 2010.

[15] Jerry Grossman. The Erdős Number Project. `http://www.oakland.edu/enp`.

[16] Stanley Milgram. The Small World Problem. *Psychology Today*, May:60—-67, 1967.

[17] Mark E. J. Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(026113):1–16, 2004.

[18] Christian Staudt and Robert Görke. A generator of dynamic clustered random graphs. Technical report, Karlsruhe Instistute of Technology, 2009.

[19] Stijn M. van Dongen. *Graph Clustering by Flow Simulation.* PhD thesis, University of Utrecht, 2000.

[20] Duncan J. Watts and Steven H. Strogatz. Collective Dynamics of 'Small-World' Networks. *Nature*, 393:440–442, 1998.

Nomenclature

$A * B$	unordered Cartesian product, set of all unordered pairs containing exactly one element from A and B
$G \sim G'$	G and G' are adjacent
$\binom{V}{2}$	all possible edges of a simple graph with node set V
$\binom{n}{k}$	binomial coefficient
$\perp$	invalid / null pointer, "no result found"
$u \sim v$	u and v are adjacent, $\{u, v\} \in E$
$\mathcal{C} = \{C_1, \ldots, C_k\}$	ground truth, current clustering
C	(ordinary) cluster
$c(v)$	cluster of node v
C_0	pseudo cluster
$\deg v$	degree of node $v \in V$, $\deg v = \deg_{\text{intra}}(v) + \deg_{\text{inter}}(v) = \lvert N(v) \rvert$
$\deg_{\text{intra}}(v), \deg_{\text{inter}}(v)$	intra-cluster degree, inter-cluster degree of node v
$\bar{E}$	set of non-edges $\binom{V}{2} \setminus E$
$\mathbb{E}[X]$	expectation value of random variable X
E	set of edges
$E(C)$	intra-cluster edges of C
$E(C_i, C_j), \bar{E}(C_i, C_j)$	set of inter-cluster edges / non-edges between clusters C_i and C_j
$E_{\text{inter}}, \bar{E}_{\text{inter}}$	set of inter-cluster edges / non-edges
$E_{\text{intra}}, \bar{E}_{\text{intra}}$	set of intra-cluster edges / non-edges
$\bar{G} = (\bar{V}, \bar{E})$	complement graph of G
$G = (V, E)$	undirected, simple graph
$G(V') = (V', E')$	node induced subgraph of $G = (V, E)$, $V' \subseteq V$
$\Gamma_{\text{ins}}, \Gamma_{\text{del}}$	cluster trees for insertion and deletion
k	number of clusters, $\lvert \mathcal{C} \rvert$
$m(C)$	number of intra-cluster edges of C

$m(C_i, C_j), \bar{m}(C_i, C_j)$	number of inter-cluster edges, inter-cluster non-edges between clusters C_i and C_j
$m, \bar{m}$	number of edges, number of non-edges
$m_{\text{inter}}, \bar{m}_{\text{inter}}$	number of inter-cluster edges / non-edges
$m_{\text{intra}}, \bar{m}_{\text{intra}}$	number of intra-cluster edges / non-edges
n	number of nodes, $\|V\|$
$n(C)$	number of nodes contained in cluster C, $\|C\|$
$N(v)$	neighborhood of node v, $N(v) = N_{\text{intra}}(v) \cup N_{\text{inter}}(v)$
$N_{\text{intra}}(v), N_{\text{inter}}(v)$	intra-cluster neighborhood, inter-cluster neighborhood of node v
$\mathcal{O}(g)$	the class / set of all functions growing asymptotically no faster than g
$p_{\text{in}}(C)$	intra-cluster edge probability for cluster C
p_{out}	intra-cluster edge probability
$p(\alpha)$	probability of event α
$P(S, p)$	probability mass of set S weighted with function $p : S \to [0..1]$
$p(u, v)$	edge probability of $e = \{u, v\}$
$\mathcal{R}$	reference clustering
$\Theta(g)$	the class / set of all functions growing asymptotically as fast as g
$T(u), \bar{T}(u)$	target tree of deletion / insertion belonging to node u
$T, \bar{T}$	source tree of deletion / insertion
V	set of nodes
$\omega_{\text{ins}}(C), \omega_{\text{del}}(C)$	insertion and deletion weight of cluster C