Università degli Studi di Padova

Department of Information Engineering

Master Degree in Computer Engineering

Identifying tumor clones with noisy itemset mining

Supervisor
Fabio Vandin
Università degli Studi di Padova

Co-supervisor
Leonardo Pellegrina
Università degli Studi di Padova

Candidate
Samuele Benfatti

Academic year 2021-2022

Ad Anna

Abstract

Sequencing the genome of single cells allows for a better understanding of cellular populations, and provides interesting insights over the composition and phylogeny of tumor samples. Unfortunately, single-cell sequencing technologies present some challenges; in particular, they often suffer from high sparsity and low signal-to-noise ratio of their output. This problem is particularly pronounced when single-nucleotide variations (SNVs) are considered, due to the high probability of false negatives.

In this work we propose a novel approach based on noisy itemset mining that allows the recovery of the clonal composition of a tumor from SNV single-cell data. An extensive theoretical foundation is presented to justify the chosen framework, and to highlight the relationship between noisy itemsets and clones with their mutations. This allows the development of a proper score, which in expectation is provably capable of distinguishing between correct clones and random groups of cells. Such score is then improved thanks to theoretical and practical considerations, and serves as basis for three different algorithms.

A comprehensive experimental section completes this work, where the efficiency, the scalability and the accuracy of the proposed techniques are evaluated on synthetic datasets and compared with a state-of-the-art algorithm. In the end, some tests are performed on synthetic data generated using realistic parameters derived from actual sequencing data; in such conditions, the proposed approach proved to outperform the state-of-the-art.

Sommario

Il sequenziamento del genoma di singole cellule permette una migliore comprensione delle popolazioni di cellule, e fornisce informazioni interessanti sulla composizione e la filogenesi di campioni tumorali. Purtroppo, le tecnologie di sequenziamento a singola cellula presentano alcune sfide; in particolare, spesso i risultati sono affetti da un'elevata sparsità e da un basso rapporto segnale-rumore. Questo problema è particolarmente pronunciato quando vengono considerate le varianti a singolo nucleotide (SNV), a causa dell'alta probabilità di registrare falsi negativi.

In questo lavoro proponiamo un nuovo approccio basato sul *noisy itemset mining* che permette la ricostruzione della composizione clonale di un tumore a partire da sequenze di SNV da singole cellule. Presenteremo un'ampia fondazione teorica per giustificare il *framework* scelto, nonché per evidenziare la relazione tra *noisy itemset* e cloni. Ciò permette lo sviluppo di un'adeguata funzione obiettivo, la quale dimostrabilmente permette di distinguere in aspettazione tra cloni corretti e gruppi casuali di cellule. Tale funzione è poi migliorata grazie a considerazioni teoriche e pratiche, e serve da base per la definizione di tre differenti algoritmi.

Un'estesa sezione sperimentale completa il lavoro; qui l'efficienza, la scalabilità e l'accuratezza delle tecniche proposte sono valutate su dati sintetici, e confrontate con un algoritmo allo stato dell'arte. Per concludere, effettueremo alcune prove su dati sintetici generati a partire da parametri realistici derivati da effettivi dati di sequenziamento; in queste condizioni, gli algoritmi proposti hanno dimostrato di superare lo stato dell'arte.

Contents

List of Figures

Chapter 1

Introduction

In this Chapter we introduce the task of interest and the underlying data model, along with the necessary notation. In particular, Section 1.1 describes and motivates the considered task; Section 1.2 introduced the model, along with a brief description of a state-of-the-art algorithm, while Section 1.3 provides an overview over related works. Finally, Section 1.4 presents the outline of this work.

1.1 Introduction to the task

In recent years the interest for single-cell DNA sequencing technologies has grown considerably [1]. Such technologies typically offer low sequencing coverage; hence, this induced an increasing need for new algorithms capable of overcoming the high sparsity and low signal to noise ratio of input data.

An example application of single-cell sequencing is the study of the clonal composition of tumor samples: suppose that we have sequenced m cells, and n mutations were identified; then, we are interested in partitioning the m cells into groups (*clones*) such that all cells in the same clone share the same mutations. If the mutations we are considering involve a large number of bases, such as with copy-number aberrations (CNAs), then it is reasonable to assume that, despite having low coverage, a mutation will nearly always be detected if present; the task of identifying the clonal composition of the sample is then almost as easy as grouping together all cells with the same mutations, as per task definition.

The problem is quite different if we consider single-nucleotide variations (SNVs) instead: in this case, due to the low coverage of available technologies, several mutations can get unnoticed even when present. The consequence is that we can recover all (or most) of the mutations in a cell only by merging data from the cells in the same clone; at the same time, however, it is very difficult to correctly identify the real clones, since checking for shared mutations requires the knowledge of the mutation profile of the considered cells, in a vicious circle.

1.2 Data model and SBMClone

A state of the art algorithm to address the second problem presented in Section 1.1 is SBMClone, proposed by Myers, Zaccaria, and Raphael [2]. To better understand it, we first need to consider the underlying data model: suppose that the sequencing data is represented through an $m \times n$ mutation matrix $D = [d_{ij}]$. For each entry, we have that $d_{ij} = 1$ if mutation j was detected in cell i, and $d_{ij} = ?$ otherwise.

This reflects the fact that, ideally, if a sequencing read contains a given mutation then we can assume that the probability of the cell actually containing such mutation is reasonably high, being limited only by the probability of having a sequencing error. In other words, the probability of having a false positive is relatively low; therefore, we can insert a one in the corresponding entry of the matrix.

Conversely, if we don't find the given mutation we cannot be sure that the mutation is actually not present; for example the mutation may only be present in one of the two copies of the diploid regions of human genome, and we may have covered only the other one. Such uncertainty should be represented by inserting a "?" in the matrix; in practice, for ease of notation a zero can be used instead. This is perfectly equivalent, as long as we recall that the actual meaning of the symbol is not the absence of a mutation, but the absence of knowledge about such mutation.

Our task is then equivalent to finding a partition of D into blocks such that each block as either an high or a low proportion of ones; in the first case, the rows of the block represent a clone, while the columns represent its mutations; in the second case we instead know that the considered group of mutations is not present in the current cells.

Such structure is modelled in SBMClone via a *stochastic block model* (SBM). SBMs are generative models; broadly speaking, suppose we have a set S of items and a relation R over a subset of $S \times S$. The items of interest are divided into disjoint subsets $S_1, S_2, \ldots, S_k$; each pair (S_i, S_j) is a *block*.

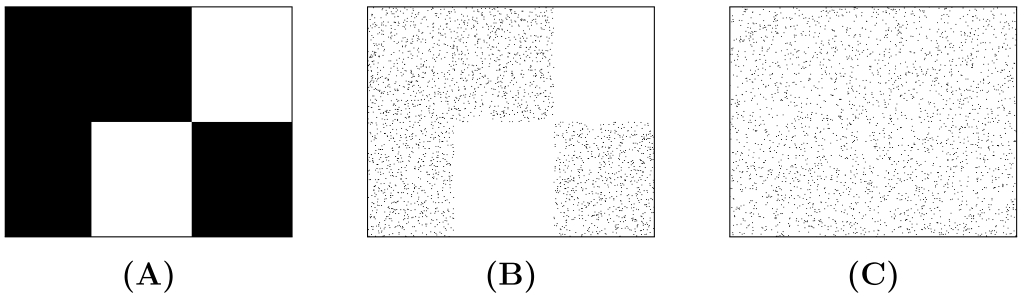

Figure 1.1: **(A)** a small mutation matrix without noise. Two clones (on the rows) and three clusters of mutations (on the columns) are present, for a total of 6 blocks. **(B)** the same matrix after applying destructive noise. **(C)** in real-world data the underlying structure is hidden: mutations and cells are not sorted in a meaningful way.

Then, a symmetric $k \times k$ matrix of probabilities $P = [p_{ij}]$ is defined over the blocks; each entry p_{ij} represents the chance that $(a, b) \in R$ for any two elements a, b such that $a \in S_i$ and $b \in S_j$.

In the considered case, each block is composed of a clone and a set of mutations; hence, the probability of correctly identifying the presence or absence of a certain mutation in a cell is the one associated to the corresponding block. Therefore, D can be obtained by sampling from the ground truth X according to the distribution we just defined, where $X = [x_{ij}]$ is a binary matrix that represents all clones and groups of mutations before the introduction of noise. In other words, all mutations of each cell are correctly marked as present with a one, and all mutations not present in each cell are correctly marked as absent with a zero; see Figure 1.1 **(A)**.

In particular, slightly changing the notation from [2], we correctly read a one with probability $1 - p > 0$ if the considered cell contains the given mutation (i. e., if $x_{ij} = 1$); if instead cell i does not contain mutation j (i. e., if $x_{ij} = 0$) we correctly read a zero with probability $1 - q$. In practice, $q = 0$ is imposed; this is consistent with the assumption of having a negligible fraction of false positives, as previously described. For $p = 0.995$, which is a value typically used in [2], this could result in something like Figure 1.1 **(B)**. Notice however that we do not know the underlying block structure of our data, since it is exactly what we are trying to reconstruct; hence, real-life data would appear more "scrambled", like in Figure 1.1 **(C)**.

Once the described data model is fixed, a good partitioning of cells and mutations can be recovered from D through an SBM inference algorithm, whose goal is to find the block matrix and the block probabilities that most likely generated our input matrix. This is exactly the approach followed by SBMClone; in particular, it uses the algorithm described in [3, 4]. Notice however that such algorithm works for general, per-block probabilities. On the other hand, in our case we are imposing the assumption that the only parameter is p, describing the probability of having a wrong read whenever a given clone contains a certain set of mutations; all other blocks are assumed to contain only zeros. Can we exploit such structure to reach better reconstruction capabilities?

1.3 Related work

As mentioned in Section 1.1, the interest for single-cell DNA sequencing technologies is continuously growing; this reflects on a quite large literature production on the subject. Several studies focus on the applications of such technologies; in particular, there is an active research trying to reconstruct the evolution of tumors and cancers by analyzing the clonal composition of cellular samples [5, 6, 7, 8].

The available algorithms to reconstruct such clones follow a wide variety of different approaches. From an high level point of view, they can roughly be divided into two categories: the ones focusing on single nucleotide variations and the ones focusing on copy number aberrations; some examples of the latter category are [9], [10] and [11].

However, there are also some algorithms where both kind of mutations are explicitly considered, so as to take advantage of the peculiarities of both of them; for example, see [12] and [13]. Another interesting variation over the basic approach can be found in [14], where single-cell sequencing data is integrated with bulk sequencing data.

As for single nucleotide variations, the most common approach involves the application of various flavours of Bayesian inference through Markov Chain Monte Carlo (MCMC) methods; some examples include [15], [16], [17], [18], [19] and SBMClone itself [2].

Other types of probabilistic approaches explored in the literature include max-likelihood [20], expectation-maximization [21], mean-field variational inference on mixture models [22] and simulated annealing [23].

Several works impose a specific evolutionary model to simplify the problem; for example, [24], [23] and [25] utilize the Dollo-k parsimony model, where each mutation can be gained only once and lost at most k times.

Finally, yet another different option consists in performing principal component analysis of the mutation matrix, which is then followed by either Louvain–Jaccard clustering [26] or k-means clustering [27].

To the best of our knowledge, no algorithm described in literature exploits noisy itemset mining to retrieve the clonal composition of single-cell mutation data.

1.4 Thesis outline

The remaining part of this work is organized as follows. First, the proposed model based on noisy itemset mining will be presented in Chapter 2; the relevant notation will be introduced, followed by some proofs that will highlight the relationship between clones, mutation clusters and itemsets.

Chapter 3 contains some interesting previous works about noisy frequent itemset mining. The need of an *ad hoc* solution will emerge; as such, we will dedicate the first Section of Chapter 4 to the development of a score that provably allows the distinction between correct clones and random groups of cells in expectation. The remaining part of the same Chapter presents two improved versions of the basic score, as well as the derivation of three clustering algorithms based on the developed theory.

Chapter 5 contains extensive experiments on synthetic datasets, evaluating the effect of varying the number of cells, the number of mutations, the false negative probability and the mutation tree. In particular, realistic parameters inspired by the actual sequencing data from [5] will be considered in Section 5.8; in such conditions, the proposed approach proved to outperform SBMClone. Finally, some concluding remarks are presented in Chapter 6.

Chapter 2

Itemset mining and clones reconstruction

In this Chapter we introduce Itemset Mining and its variants, along with the relationship between itemset and clones. In particular, noisy frequent itemsets and their notation are introduced in Section 2.1, while the reconstruction of the clonal composition from noisy itemsets is formally derived in Section 2.2.

2.1 Noisy Frequent Itemsets Mining

In the previous Chapter we highlighted the kind of structure that is present in the data of interest; to fully exploit such structure *frequent itemset mining* is a particularly suited tool. We will hence introduce the basic formalism, and then proceed to adapt such framework to the considered problem.

Let us then start from the standard definition of frequent itemsets mining: we have a set of transactions $C = \{c_1, c_2, \ldots, c_m\}$ and a set of items $V = \{v_1, v_2, \ldots, v_n\}$, whose relation is represented through an $m \times n$ binary matrix $D = [d_{ij}]$. We have that $d_{ij} = 1$ if transaction i contains the item j, and $d_{ij} = 0$ otherwise. An itemset x is defined as a subset of V; we say that a transaction c_i supports x if $d_{ij} = 1 \ \forall j \in x$. Then, we say that an itemset is frequent if the fraction of all transactions that support it is greater or equal to a threshold $min\_sup > 0$. The goal of frequent itemsets mining is to retrieve all frequent itemsets, possibly along with their support.

This model can be adapted to represent mutation data: D exactly corresponds to the mutation matrix of the same name defined in Section 1.2, C is the set of sequenced cells and V is the set of single-nucleotide variations (or, more in general, mutations) we found.

To complete the adaptation of our input data to the frequent itemset mining framework, all is left to do is to consider noise. Let us then formally introduce the following definition.

Definition 2.1.1 (Destructive noise)**.** Let D be a mutation matrix. D is affected by *destructive noise* if each entry with value one is independently turned into a zero with a certain probability $p < 1$, where p is the *strength* of the noise.

Then, if our matrix D is affected by destructive noise, *noisy frequent itemset mining* is the task of recovering all the itemsets that would be frequent if no noise was present. Notice that this task is considerably more difficult than simply mining frequent itemsets from noiseless data. We will explore in the next Section all the consequences that this increased complexity has on the recoverability of the correct clonal composition.

2.2 Reconstruction of clones and mutations

We already adapted our input data to the frequent itemset model; what we need to do now is to adapt the output. This first requires us to formally define what exactly are we looking for. We have already seen that our goal is to partition our set of cells C into disjoint clones $\mathcal{C} = \{C_1, C_2, \ldots, C_h\}$, such that all cells inside a clone share all their mutations, and for any two cells from different clones there is at least a mutation in a cell that is not present in the other one. Similarly, we also want to partition the set of mutations V into disjoint subsets $\mathcal{V} = \{V_1, V_2, \ldots, V_k\}$, such that each subset V_j contains all mutations that are exclusive to a certain set of clones, and only those mutations.

To better understand this, let us consider as an example the mutation matrix in Figure 2.1. The two clones we want to recover are clearly C_1 and C_2; as for the mutations, we want to retrieve V_1 and V_2, where V_2 contains all mutations exclusive to $\{C_2\}$ and V_1 contains all mutations exclusive to $\{C_1, C_2\}$. Obviously $V_1 \cap V_2 = \emptyset$ and, since there are no mutations exclusive to $\{C_1\}$, $V_1 \cup V_2 = V$.

Let us ignore noise for the moment. We would like each clone to support an itemset containing the clone's mutations, while at the same time we would like each itemset to correspond to a different clone. In practice this is not trivial: the problem we face is that every subset of a frequent itemset

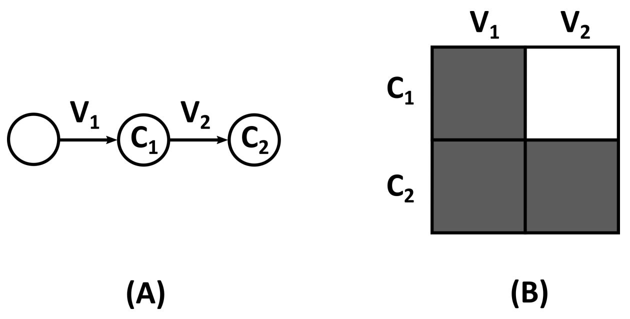

Figure 2.1: **(A)** A basic clone tree and **(B)** its corresponding mutation matrix.

must be a frequent itemset itself, and would hence be returned in output by a generic mining algorithm. In our setup this is not desirable, since we are only interested in a specific subset of all possible frequent itemsets.

We could then be tempted to look only for maximal frequent itemsets, i. e. frequent itemsets that are not included in any frequent superset. But then, returning to our example, we would only find $V_1 \cup V_2 = V$, supported by C_2. This strategy is too much restrictive, and hence does not work in general.

A slightly better option is to consider frequent closed itemsets, i. e. frequent itemsets that are not included in any frequent superset *having the same support*. Returning to our example, this way we can easily find V_1, supported by $C_1 \cup C_2$. This is actually a step in the right direction, allowing us to prove the following results:

Proposition 2.2.1. *Assume no noise is present in D. Then, for a proper choice of min_sup each clone $C_i \subseteq C$ supports a uniquely defined closed frequent itemset.*

Proof. Let C^* be the clone with the fewest cells. Let us choose a value for *min_sup* such that

$$\frac{|C^*|}{m} \geq \mathit{min\_sup} > 0.$$

Notice that by construction it holds true that for any clone C_i

$$\frac{|C_i|}{m} \geq \frac{|C^*|}{m} \geq \mathit{min\_sup} > 0.$$

By definition, all cells included in C_i share precisely the same set of mutations; let x be such set. It is then easy to verify that by construction each cell in C_i supports all and only the items in x; this means that the fraction of transactions that support x is indeed greater or equal to min_sup, thus making x a frequent itemset.

We can then notice that x must indeed be a closed itemset, since the cells in C_i cannot support any additional mutation; this also means that we cannot expand x in any way. Symmetrically, removing any mutation from x would make it not closed; therefore, x is uniquely defined. □

Proposition 2.2.2. *Let no noise be present in D. Let $x \subseteq V$ be a closed frequent itemset supported by $C_x \subseteq C$. Then, there is a set of cells C_x^* that corresponds to a (possibly empty) clone.*

Proof. Given a closed frequent itemset $x \subseteq V$, the set of transactions supporting it is uniquely defined as

$$C_x \overset{\text{def}}{=} \{c_i \in C \text{ s.t. } d_{ij} = 1 \ \forall j \in x\}.$$

Such set must be non-empty, since for x to be frequent there must be at least a transaction supporting it. By construction, C_x contains all and only the cells with all the mutations $v_j \in x$. We can then define the subset of all the cells $C_x^* \subseteq C_x$ that contain only such mutations. Notice that it may happen that such subset is empty. In any case, all cells in C_x^* share exactly the same set of mutations; as such, they correspond to a (possibly empty) clone. □

In other words, if each block of mutations that appeared during the tumor evolution actually corresponds to a specific group of cells, then we can identify a one-to-one correspondence between closed frequent itemsets and clones. Unfortunately, it is quite clear that this is too strict a requirement; to show why we just need to refer to the example from Figure 2.2.

This time, four closed frequent itemsets can be identified: V_1 (supported by $C_1 \cup C_2 \cup C_3$), $V_1 \cup V_2 \cup V_3$ (supported by C_2), $V_1 \cup V_2 \cup V_4$ (supported by C_3) and $V_1 \cup V_2$ (supported by $C_2 \cup C_3$). While each of the first three itemsets corresponds to the set of all mutations of a different clone (respectively C_1, C_2 and C_3) the fourth one does not correspond to any clone. This is due to the fact that the node that in Figure 2.2 **(A)** corresponds to the child of C_1 does not actually contain any cell. Notice that this is actually a quite important scenario in practice: for example, it corresponds to the situation in which a treatment lead to the extinction of a specific tumor clone, but the artificial selective pressure introduced by the treatment itself induced the appearance of one or more new therapeutically resistant clones.

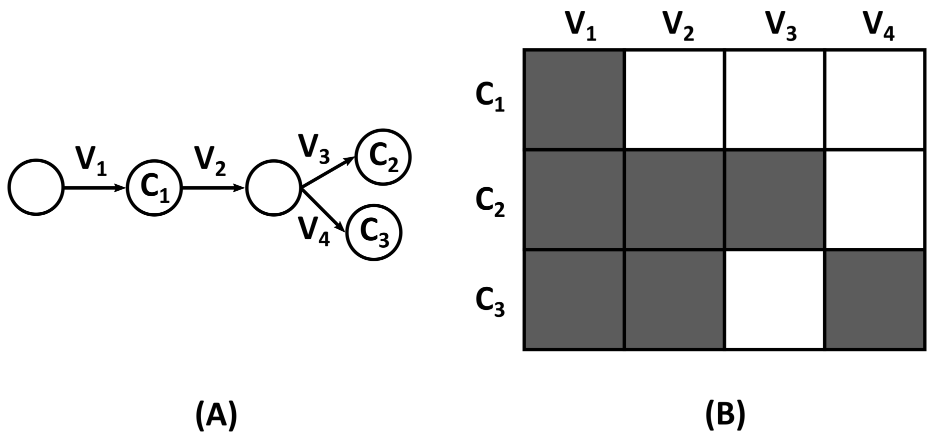

Figure 2.2: **(A)** A more complex clone tree and **(B)** its corresponding mutation matrix.

To actually find a nice and robust correspondence between itemsets and clones, we need to introduce a new type of frequent itemset:

Definition 2.2.1 (Exact support)**.** A transaction $c_i \in C$ *exactly supports* the itemset $x \subseteq V$ if $d_{ij} = 1 \; \forall j \in x \; \wedge \; d_{ij} = 0 \; \forall j \notin x$.

Definition 2.2.2 (Exact frequent itemset)**.** An itemset $x \subseteq V$ is an *exact frequent itemset* if the fraction of all transactions that exactly support it is greater than a threshold $min\_sup > 0$.

Notice how according to this definition all frequent exact itemsets are closed itemsets: if a transaction exactly supports an itemset, it cannot support any bigger itemset at the same time. Conversely, not all closed itemsets are exact: in the example from Figure 2.2, $V_1 \cup V_2$ is closed but not exact. This means that exact itemsets are indeed more restrictive; we can then actually prove that a one-to-one correspondence between frequent exact itemsets and clones holds in general:

Proposition 2.2.3. *Assume no noise is present in D. Then, for a proper choice of min_sup each clone $C_i \subseteq C$ supports a uniquely defined exact frequent itemset.*

Proof. Trivially, we can adapt the proof of Proposition 2.2.1 by noticing that the cells in C_i are actually exactly supporting x; as such, the resulting itemsets are not simply closed, but are actually exact. $\square$

Proposition 2.2.4. *Let no noise be present in D. Let $x \subseteq V$ be a frequent exact itemset supported by $C_x \subseteq C$. Then, C_x is a unique, non-empty clone.*

Proof. Given a frequent exact itemset $x \subseteq V$, the set of transactions exactly supporting it is uniquely defined as

$$C_x \overset{\text{def}}{=} \{c_i \in C \text{ s.t. } d_{ij} = 1 \ \forall j \in x \ \wedge \ d_{ij} = 0 \ \forall j \notin x\}.$$

Such set must be non-empty, since for x to be frequent there must be at least a transaction supporting it. By construction, C_x contains all and only the cells with all and only the mutations $v_j \in x$. By definition, this means that C_x is a clone. □

Corollary 2.2.4.1. *Different exact itemsets are supported by different, non-intersecting clones.*

Proof. Let $x_1, \ x_2 \subset V$ be two closed itemsets, $x_1 \neq x_2$. Thanks to Proposition 2.2.4 we know that the support of x_i is a unique, non-empty clone $C_i \subset C$ whose cells contain all and only mutations from x_i, $i = 1, 2$. By definition, a clone only contains cells with exactly the same mutations; since $x_1 \neq x_2$, no cell from C_1 can be included in C_2, and no cell from C_2 can be included in C_1. Therefore, it holds true that $C_1 \cap C_2 = \emptyset$ and, since both clones must be non-empty, $C_1 \neq C_2$. □

All in all, what we have proved is that we can always recover $\mathcal{C}$ from frequent exact itemsets, as long as no noise is present; at the same time, no spurious frequent exact itemset is returned in output. Still, we cannot recover $\mathcal{V}$ yet: so far we have grouped all mutations of a clone in a single set; returning once again to our example from Figure 2.1, this means that we would not retrieve V_2, as the only exact itemset supported by C_2 is V.

Luckily this problem can easily be solved in the general case by post-processing the itemsets: we proved that all transactions that exactly support an itemset define a single clone; and that at the same time if we consider different itemsets the corresponding clones are different, too. Therefore, we can label each mutation with the set of itemsets (and hence clones) that contain it; then, let us cluster together all mutations with the same label. This results in a partition $\{V_1, V_2, \ldots, V_k\}$ of V such that each subset V_j will contain a mutation if and only if such mutation is exclusive to the set of clones which labeled V_j: this is precisely what we required. In our example, all mutations in V_1 would be labeled $\{C_1, C_2\}$, while all mutations in V_2

would be labeled $\{C_2\}$, thus enabling us to correctly reconstruct the two blocks of mutations we are interested in.

This clearly holds true only if the returned closed itemsets are the correct ones; we have already seen that this is indeed the case when no noise is present. But what about the real, noisy setup?

Unfortunately, it is easy to verify that in the general case the introduction of noise may invalidate Proposition 2.2.4: for example, referring once again to Figure 2.1, a cell from clone C_2 may lose due to noise all mutations from V_2. Then, there would be no way to distinguish such cell from the ones composing clone C_1; hence, we cannot guarantee that all transactions exactly supporting an itemset correspond to a true clone.

More in general, it can be proven that the introduction of noise leads to breakage of large frequent itemsets into subsets of logarithmic size, even for moderate values of p (see [28, 29]). Hence, the supports of the returned exact frequent itemsets may not only include spurious cells, as just observed, but may also include only a fraction of the original cells.

To mitigate such loss of recoverability we need to introduce more sophisticate techniques, and several options have already been proposed by literature; see Chapter 3 for more details. Nevertheless, we can still provide a useful guideline by proving that a weaker version of Proposition 2.2.3 holds:

Proposition 2.2.5. *Let D be a mutation matrix affected by destructive noise. Then, for each clone $C_x \subseteq C$ originally present in the data there is a uniquely defined itemset x that contains the maximum number of mutations originally present in C_x, while guaranteeing that no additional mutation is wrongly introduced.*

Proof. Since C_x is a true clone, even in the presence of destructive noise all cells included in it share only (but not all) the same mutations. Let the set of such mutations be x; formally, $x = \{j \in V \text{ s.t. } \exists\ d_{ij} = 1 \ \wedge \ c_i \in C_x\}$. Notice that destructive noise cannot introduce a new mutation in a cell; hence, by construction each mutation contained in x was for sure present in the original clone. Obviously, if we remove any item from x we are left with a smaller set. At the same time, if we add to x any other mutation $j \in V \setminus x$ we have no proof that such mutation was indeed originally present in clone C_x, since by construction $\nexists\ d_{ij} = 1$ s.t. $j \notin x \ \wedge \ c_i \in C_x$. Hence, x is the only itemset respecting the required terms. $\square$

In other words, if we hope to correctly recover the original clones and their mutations we still need to include in the itemsets *all* and *only* the mutations present in the transactions that support them, even tough we now have fewer guarantees about the correctness of the resulting itemsets. This means that we will need a method to heuristically distinguish between “correct” frequent itemsets and “wrong” ones; this will be one of the key ideas behind the algorithms proposed in Chapter 4.

Chapter 3

Previous works

The problem of mining frequent itemsets from noisy data has been extensively studied in literature. In this Chapter we will present a brief overview of some interesting results and algorithms; notice that whenever appropriate the notation of the original papers will be modified to follow the one used in this work. In particular, the relevant literature is presented in Sections 3.1 through 3.5, while Section 3.6 completes the Chapter with some general considerations about the available techniques and their limitations.

3.1 Error-Tolerant Itemset (ETI)

One of the first appearances of the concept of noisy itemset mining is in the work of C. Yang, U. Fayyad and P. S. Bradley [30]. In particular, they notice that standard frequent itemset mining algorithms tend to fail on relational databases where a relatively small amount of noise is present. Therefore, a generalization of the definition of supporting transaction is introduced, allowing for the presence of a fixed fraction of errors in each transaction. More formally, they define a *(strong) Error-Tolerant Itemset* (ETI) for a given error tolerance $\varepsilon > 0$ as any itemset x such that in the database D there exist at least $min\_sup \cdot m$ transactions in which the fraction of ones over the elements of the itemset is not less than $1 - \varepsilon$. Notice how if we let $\varepsilon = 0$ instead then this definition is equivalent to the one of a normal frequent itemset.

To show that ETIs are not just random artifacts that appear from noisy data, the authors proceed by proving that the probability of finding an ETI inside a $m \times n$ binary matrix where entries are i.i.d. Bernoulli variables rapidly vanishes as ε diminishes and min_sup grows.

The next problem lies in finding all ETIs in a dataset. Unlike frequent itemsets, ETIs do not follow the anti-monotone property; as such, following the same approach of the Apriori algorithm would not work. The solution is to further relax the definition of ETI, leading to *weak ETI*s. In particular, the zeros are now allowed to be freely distributed between the supporting transactions; in other words, some supporting transactions may contain less than $(1-\varepsilon) \cdot |x|$ items from x, as long as the average fraction of ones among all considered transactions is still at least $1 - \varepsilon$.

It is then proved that if x is a weak ETI then it always includes an item j^* such that the itemset $x' = x \setminus \{j^*\}$ is still a weak ETI. Exploiting this property, itemsets can be incrementally expanded in a bottom-up fashion, resulting in an exhaustive algorithm similar to Apriori. Once weak ETIs are identified, it is then easy to filter out the ones not corresponding to a strong ETI. However, this still results in a runtime exponential in the size of the considered itemsets; therefore, a greedy approximation is introduced.

The main idea is to use some heuristics to restrict how the algorithm tries to expand the current itemsets. However, the introduced heuristics may lead to the loss of some ETIs; to mitigate this problem as much as possible, an iterative sampling and validation scheme is introduced. In particular, multiple rounds of the main algorithm are performed; each time a round is completed, the database is reduced by removing all transactions that supported an itemset in the previous rounds, repeating the process until no more new ETIs can be found. Furthermore, the heuristic conditions which define when an itemset is taken into consideration for expansion are relaxed starting from the second round, with the effect of modifying the tradeoff between the probability of missing an itemset and the time requirements. To reduce complexity even further, a sampling approach is proposed: at each round the algorithm only runs on a smaller sampled database; then, the retrieved itemsets are validated on a different sample of the database, so as to reduce the presence of spurious itemsets.

To evaluate the performance of the resulting algorithm some tests are performed on both synthetic and real-world datasets. Starting from the latter category, the main applications of the algorithm that are explored in [30] are clustering high-dimensional data, query selectivity estimation and collaborative filtering. The first task is the one most closely related to our problem: ETIs are used to initialize some clusters, which are then refined using expectation-maximization clustering algorithms. Experimentally, the considered clustering algorithms tend to retrieve only a subset of the actual clusters when directly applied to high-dimensional data. On the other hand, using ETIs as seeds greatly improve the recovery of all clusters, with the initial ETIs often corresponding directly to the final, correct clusters.

As for synthetic data, several different matrices are generated by changing the relevant parameters. The noise model utilized imposes $p > q > 0$; in other words, it is assumed that there can be both false positives and false negatives, despite the former being more rare. Notice how this model differs from the one introduced in Chapters 1 and 2, where only false negatives are allowed. To evaluate performance, both missing and spurious ETIs are considered. The first group includes all itemsets that were frequent in the original noiseless data and that are not identified by the algorithm; the second group includes all itemsets returned by the algorithm that were not actually frequent in the original data. Experimentally, the proposed algorithm is capable of retrieving all and only the correct itemsets, provided some conditions are met. In particular, as the overlap between different itemsets (i. e., the fraction of items included in all the considered itemsets) increases, the performance of the algorithm gradually degrades. According to [30], this degradation is acceptable in the real-world applications considered by the authors. However, this may actually be a problem when looking for clones and mutation cluster, since in this case entire itemsets of interest can be subsets of bigger itemsets; for example, this is exactly what happens in Figure 2.1. What's more, all tests were performed using noise levels around $p = 0.075$; since for mutation data we can expect a noise level well above $p = 0.9$, a further degradation in performance levels can probably be expected.

The last potential issue regards runtimes. The asymptotic complexity of the considered algorithm is $O(cdh^2)$, where c is the number of ETIs, d is the average number of items in an ETI, h is the number of items whose global count of ones is at least $min\_sup \cdot m \cdot (1-\varepsilon)/\lambda$, and $\lambda \geq 1$ is a parameter used by the previously mentioned heuristics. This corresponds to a worst-case complexity which is cubic in $|V|$ whenever most items are actually frequent and most transaction include a substantial fraction of all items. While this does not happen in practice in the tests performed in [30], this conditions would probably apply to mutation data, leading to unfeasible runtimes.

3.2 Approximate Frequent Itemset (AFI)

A first expansion of the ETI model is presented by J. Liu *et al.* [31]. Once again, they focus on relational databases; however, they also recognize that noisy itemset mining presents broader applications, such as subspace clustering and building classifiers. In these cases, the associations that we want to discover are not only between items, but also between items and transactions. This is actually what we need in order to solve the clonal composition problem, since we want to retrieve both a clustering of mutations and a corresponding clustering of cells. Unfortunately, in [31] it is noticed that when noise is present finding the transactions that support a noisy itemset is not trivial, even if the itemset itself is already known.

The second limitation discussed specifically involves the definition of ETIs; in particular, the fact that they place no restriction over the distribution across columns of the allowed zeros. As such, nothing prevents the presence of a column without ones; the corresponding item would then be considered part of the itemset, even tough this result would probably be spurious.

To solve this problem, *Approximate Frequent Itemsets* (AFIs) are introduced. Their definition is exactly the same of strong ETIs, but with an additional constraint over columns: for each item, the fraction of the considered transactions that support it should be at least $1 - \varepsilon_c$. Notice how this constraint is symmetrical to the one already present in the definition of ETIs.

The first approach presented to identify AFIs is simply a brute-force search: first, all ETIs are retrieved using the algorithm from [30]; then, the result is filtered by enforcing the additional column-wise constraint. Experimental evaluation on synthetic data shows that enforcing the AFI constraints consistently results in lower spuriousness, while at the same time yielding equal or better recoverability. However, the resulting runtime is quite high, mainly due to the need of identifying all ETIs.

A better option is presented by J. Liu *et al.* in [28]. The key idea is once again to identify an anti-monotone property for AFIs, enabling the adoption of a pruned breadth-first search similar to the one used by the Apriori algorithm. This leads to the introduction of *0/1 Extensions*: itemsets are sorted according to their cardinality, and each time the cardinality increases by one the maximal number of zeros allowed in each transaction that satisfies the AFI constraints can either remain the same or increase by one. In the first case, any transaction that does not support a subset of an itemset will not support the itemset itself; in the second one, any transaction supporting a subset also supports its superset. Taken together, this two properties enable the identification of the support of any itemset just by looking at the supports of its subsets. This fact, along with better pruning strategies, allows AFI to outperform ETI time-wise by a large margin.

As for result quality, once again the algorithm based on AFIs shows less spurious results, while maintaining a comparable recoverability of the correct frequent itemsets. Notice however that noise is modelled by flipping uniformly at random some entries of the binary matrix D, which corresponds to fixing $q = p > 0$; this, together with the usage of no noise strength higher than $p = 0.2$, may lead to different results in our setup. Furthermore, this means that the considered algorithm cannot take directly advantage from our knowledge that $q = 0$.

3.3 Analysis by Sun and Nobel

A theoretical analysis of the statistical properties of noisy binary matrices can be found in the work by X. Sun and A. B. Nobel [29]. Their focus is on the recoverability of submatrices of ones when each entry is disturbed by independent Bernoulli noise, i. e. when $q = p > 0$. While once again this is not the model we are assuming best represents sequencing data, it can nevertheless be interesting to consider it due to its close relationship with the work presented in Section 3.2.

Several results are presented; in this context, we are particularly interested in two of them. The first one states that noise breaks submatrices of ones into fragments of logarithmic size; more formally, given any sequence of $n \times n$ binary matrices where each entry is independently flipped with probability $0 < p < 0.5$, and given any $\varepsilon > 0$, as n grows then eventually almost surely the largest submatrix of ones that remains after noise is applied has a size between $(2 - \varepsilon)\log_{\frac{1}{p}}(n)$ and $2\log_{\frac{1}{1-p}}(n)$. As mentioned in Chapter 2, this implies that standard frequent itemset mining algorithms are bound to fail on noisy data, no matter how weak the noise is.

The second result directly involves AFI; in particular, suppose that a single $l_n \times l_n$ matrix of ones is embedded into a larger $n \times n$ matrix whose all other entries are zeros, and as previously described assume each entry is independently flipped with probability p. Then, as $n \longrightarrow \infty$ eventually almost surely the biggest AFI and its support correspond to the originally embedded matrix of ones, as long as $\frac{l_n}{\ln n} \longrightarrow \infty$. All in all, despite the differences in the utilized data model, these results show that noisy itemset mining techniques can recover matrices of ones even when noise is present, and thus can potentially be adapted to the problem of recovering the clonal composition from single-nucleotide mutation data.

3.4 AC-Close

A further improvement over ETI and AFI is AC-Close [32], proposed by H. Cheng, P. S. Yu and J. Han. The previously described approaches explore and return all maximal itemsets found; while this may be needed in some contexts, it greatly increases runtimes. Furthermore, we may not be interested in the complete collection of itemsets; this is actually our case, as discussed in Chapter 2.

To solve this problem, AC-Close only considers closed itemsets. This result is achieved by exploiting *core patterns*, i. e. the itemsets that are supported according to the standard definition by at least $\alpha \cdot min\_sup$ transactions, for $\alpha \in [0, 1]$. In particular, AC-Close starts by finding all core patterns using a standard frequent itemset mining algorithm; then, a top-down approach is followed: starting from the largest itemsets, the same constraints used in AFI are efficiently enforced by considering all subsets of the current itemset x that contain at least $|x| \cdot (1 - \varepsilon)$ items. This way, all transactions that according to the standard definition support such subsets are for sure supporting x, too. If the union of all the considered transaction has cardinality greater or equal to $|C| \cdot min\_sup$, then the constraint over the frequency of the itemset and the one over the maximum number of zeros in each transaction are automatically satisfied. Therefore, the algorithm can proceed by enforcing the constraint over the distribution of zeros across the items, as well as the closeness constraint. All in all, this approach enables AC-Close to reach better pruning capabilities and, as a consequence, faster runtimes.

This was experimentally confirmed, with evaluations on synthetic data showing a consistent and substantial increase in performance with respect to AFI, while returning a similar set of itemsets; further tests from the same authors of AC-Close can be found in [33], supporting once again the same conclusions. However, all tests are performed using a relatively small average number of items per transaction; and unfortunately, in [32] it is noted that both AFI and AC-Close present a runtime exponential in the average transaction length. Notice that for mutation data we expect the cardinality of the biggest itemset to be approximately of the same order of magnitude as $|V|$; this could probably lead to unfeasible runtimes.

As for output structure, notice that while we do not exactly need closed itemset to reconstruct the clonal composition of our mutation data, this could still be a first step in the right direction, especially considering the fact that the supporting transactions can easily be returned in output by AC-Close, and it is therefore easy to filter out non-exact itemsets. Unfortunately, the proposed approach is not applicable to our problem, since it is easy to verify that the probability that at least a cell from each clone contains all of its mutations rapidly vanishes as the number of mutations and the noise strength increase. Therefore, it is unlikely that any of the itemsets we are actually interested in corresponds to a core pattern, no matter the value of α. Furthermore, both the assumption that $q = 0$ and the knowledge that in our setup groups of transactions supporting different itemsets must be non-intersecting are not exploited by AC-close, due to its general-purpose nature.

3.5 HANCIM

A quite different approach is proposed by K. Mouhoubi, L. Letocart and C. Rouveirol [34]. The focus is on overcoming the limitations of competing algorithms on large noisy datasets, especially regarding the number of uninteresting results, the execution time and the difficulties with overlapping itemsets.

The proposed algorithm, HANCIM, follows an heuristic approach based on max-flow/min-cut graphs algorithms: the binary matrix D is interpreted as a bipartite graph, where one partition represents items and the other one transactions; once again, it is assumed that entries are independently flipped with probability $q = p > 0$. HANCIM then iteratively chooses a seed itemset, which is then grown to encompass a dense region of the graph. To find appropriate seeds four different heuristic strategies are presented and tested, with the one offering the best tradeoff between spuriousness and recoverability being chosen for the final algorithm.

The growing part of the algorithm works by considering the bipartite subgraph incident on the seed itemset; a minimal s-t cut is then performed, where weights are a function of the degrees of the destination and the source nodes. This results in the identification of some transactions strongly associated with the itemset; the procedure can then be repeated starting from the bipartite graph associated with such transactions. The process is hence iterated back and forth, until no further extension is possible.

Therefore, HANCIM returns a single final itemset for each initial seed. Notice that this does not in general correspond to the complete set of frequent itemsets which are present in D; this is an explicit design choice, since the goal is to efficiently find a small number of dense regions, thus providing an easily understandable output. However, this may be a limitation in our setup, where we are interested in completely partitioning both C and V.

As for experimental evaluation, no direct comparison with the previously mentioned methods was presented. However, tests on synthetic datasets showcase an higher time efficiency, while still providing high recoverability and low spuriousness. Furthermore, some tests were performed using overlapping itemsets, like the ones we may encounter in our setup as argued in Section 3.1. The results indicate that strongly intersecting regions may still be merged in the final output, but recoverability in the presence of overlap is in general much higher than the one displayed by ETI.

3.6 General considerations

What we have seen so far highlights how noisy frequent itemset mining seems to be a good framework to model our problem, both from a theoretical standpoint and from a practical one, with several available algorithms that do work well in practice in certain setups.

However, none of the considered alternatives perfectly fits our problem. To better understand why, we should consider the data models and the parameter space utilized by each algorithm; we can refer to Table 3.1 for a summary. The first common problematic aspect is that all the considered approaches assume that $q > 0$; as such, they do not explicitly exploit the absence of false positives, thus providing no additional benefit in this direction with respect to the algorithm used by SBMClone.

Secondly, no algorithm returns closed itemsets, which as described in Section 2.2 are needed to reconstruct mutation clusters. The only exception is AC-Close, which unfortunately cannot be directly applied to our problem, as previously explained.

Lastly, while the number of transactions and items span roughly the values we are interested in, the error probabilities and the itemset sizes targeted lie in a range quite different from the one required to reconstruct clonal compositions, both being at least an order of magnitude lower than needed. While this may not be a problem in practice, the experiments presented in the considered works show several hints that increasing itemsets size by one to two orders of magnitude could result in impractical runtimes for most algorithms. This is due to the fact that in general bigger itemsets are constructed by enumerating and processing all their subsets, resulting in an exponential complexity.

Similarly, increasing the noise strength to well over 0.9 could, according to projected results, reduce the reconstruction accuracy below usability; this can be reasonably expected in the general setup where $q > 0$, and once again highlights the need of fully exploiting our data model.

Moreover, source code could not be found for any of the presented algorithm. Therefore, all the factors we just discussed suggest the need for an ad hoc itemset mining algorithm; this will be the focus of the next Chapter.

	ETI	AFI	AC-Close	HANCIM
noise model	$p > q > 0$	$q = p > 0$	$q = p > 0$	$q = p > 0$
noise probability p	$p = 0.075$, $q = 10^{-4}$	$0.01 \div 0.2$	0.05	$0 \div 0.2$
retrieved itemsets	maximal	maximal	closed	maximal
transactions	$500k$	$10k \div 28k$	$20k$	1000
items	5000	100	100	50
avg. itemset size	$5 \div 8$	10	10	$5 \div 6$

Table 3.1: Comparison of the available algorithms' noise models and their explored parameter space.

Chapter 4

Proposed algorithms

The general idea behind most algorithms presented in Chapter 3 is to start from noisy frequent itemsets with a small number of items, and then expand them until a certain score is not high enough to allow for further expansion; of course, the exact score choice changes between the algorithms. This approach can be adapted to our problem, too; in particular, we need to find both a score that (hopefully) rewards better itemsets, and a way to extract some itemset that maximize such score. This Chapter starts with the first task in Section 4.1, where a score is first proposed and analyzed and then improved . The second task is discussed in Section 4.2, where three different algorithms are derived; the time complexity of such algorithms is analyzed and improved in Section 4.3. The most advanced algorithm among the proposed ones requires some sampling; Section 4.4 concludes the Chapter by presenting some theoretical considerations about the number of samples needed.

4.1 Density of an itemset

We have already highlighted the importance of choosing a proper score at the end of Section 2.2. A natural option is presented in 4.1.1; an extensive theoretical analysis follows, enabling us to understand some of its key properties. Some practical considerations allow us to refine the basic score (4.1.2), while a further improvement is proposed thanks to the mentioned theoretical analysis 4.1.3. The resulting improved score is not computable in closed form; therefore, a different approach is proposed.

4.1.1 Basic definition and its properties

Let us start from the following definition:

Definition 4.1.1 (Density). Let $x \subseteq V$ be a generic non-empty itemset. Let $S \subseteq C$ be a non-empty set of transactions. Then, the *density of* x *in* S is defined as

$$\delta(x, S) = \frac{\sum_{c_i \in S} \sum_{j \in x} d_{ij}}{|x| \cdot |S|}.$$

It is easy to verify that the numerator of $\delta(x, S)$ corresponds to the total number of times an item from x is actually part of one of the considered transactions. Similarly, its denominator is the number of items that would be part of a transaction, if all elements from S exactly supported x in the absence of destructive noise. As such, $\delta(x, S) \in [0, 1]$.

This definition works for general choices of S and x, and is quite a simple idea; as such, it already served as the foundation upon which several of the more advanced definitions presented in Chapter 3 are built. What actually differentiates the various proposed scores is the set of additional constraints that are imposed. In particular, what we really want to do here to fully exploit the structure of our model is to apply Proposition 2.2.5.

Hence, given a set of transactions $S \subseteq C$ we will always consider the itemset $x(S)$, defined as

$$x(S) \overset{\text{def}}{=} \{j \in V \text{ s.t. } \exists\ d_{ij} = 1 \ \wedge \ c_i \in S\}.$$

With a similar spirit, we can also introduce the set of mutations originally present in the considered cells, i. e.

$$x^*(S) \overset{\text{def}}{=} \{j \in V \text{ s.t. } \exists\ c_i \in S \text{ which originally contained mutation } j\}.$$

Furthermore, notice that empty transactions do not carry any information, and in practice can thus be removed from our matrix D; therefore, we are only interested in the case where $x(S) \neq \emptyset$. With this choice, some interesting results can be proved; let us start with the following lemma, needed as an intermediate step:

Lemma 4.1.1. *Let D be a mutation matrix affected by destructive noise with strength p. Let $S \subseteq C$ be a non-empty subset of a clone originally present in the data, and let $t \in \{1, 2, \dots, |x^*(S)|\}$. Then,*

$$\mathbb{E}[\delta(x(S), S) \mid |x(S)| = t] = \frac{1 - p}{1 - p^{|S|}}.$$

Proof. Thanks to Proposition 2.2.5, we know that $x(S) \subseteq x^*(S)$. Furthermore, by construction for each $j \in x^*(S)$ and for each $c_i \in S$ it holds true that $d_{ij} = 1 \Rightarrow j \in x(S)$; hence,

$$\sum_{j \in x(S)} d_{ij} = \sum_{j \in x^*(S)} d_{ij}. \tag{4.1}$$

Notice that S, t and $x^*(S)$ are constants, while $x(S)$ and d_{ij} are random variables. Let $P(t)$ be the proposition "$|x(S)| = t$". Then,

$$\begin{aligned}
\mathbb{E}[\delta(x(S), S) \mid P(t)] &= \mathbb{E}\left[\frac{\sum_{c_i \in S}\sum_{j \in x(S)} d_{ij}}{|x(S)| \cdot |S|} \,\middle|\, P(t)\right] && \text{by definition} \\
&= \mathbb{E}\left[\frac{\sum_{c_i \in S}\sum_{j \in x^*(S)} d_{ij}}{t \cdot |S|} \,\middle|\, P(t)\right] && \text{4.1, } P(t) \\
&= \frac{1}{t \cdot |S|} \sum_{c_i \in S} \sum_{j \in x^*(S)} \underbrace{\mathbb{E}\left[d_{ij} \mid P(t)\right]}_{(*)} && \text{linearity}
\end{aligned}$$

Let us focus on $(*)$:

$$\begin{aligned}
\mathbb{E}\left[d_{ij} \mid P(t)\right] &= 1 \cdot \mathbb{P}\left[d_{ij} = 1 \mid P(t)\right] + 0 \cdot \mathbb{P}\left[d_{ij} = 0 \mid P(t)\right] && \text{by definition} \\
&= \mathbb{P}\left[d_{ij} = 1 \mid P(t)\right]
\end{aligned}$$

but thanks to the law of total probability

$$\begin{aligned}
\mathbb{P}\left[d_{ij} = 1 \mid P(t)\right] &= \mathbb{P}\left[d_{ij} = 1 \mid j \in x(S) \cap P(t)\right] \cdot \mathbb{P}\left[j \in x(S) \mid P(t)\right] \\
&\quad + \mathbb{P}\left[d_{ij} = 1 \mid j \notin x(S) \cap P(t)\right] \cdot \mathbb{P}\left[j \notin x(S) \mid P(t)\right] \\
&= \underbrace{\mathbb{P}\left[d_{ij} = 1 \mid j \in x(S) \cap P(t)\right]}_{(**)} \cdot \underbrace{\mathbb{P}\left[j \in x(S) \mid P(t)\right]}_{(***)}
\end{aligned}$$

where we exploited the fact that $\mathbb{P}\left[d_{ij} = 1 \mid j \notin x(S) \cap P(t)\right] = 0$, since if $j \notin x(S)$ by construction $d_{ij} = 0 \ \forall c_i \in S$.

Going on, let $P(s, j)$ be the proposition "$|\{c_i \in S \text{ s.t. } d_{ij} = 1\}| = s$", $s \in \mathbb{N}$. Notice that if we know that $j \in x(S)$ then the value of d_{ij} does not depend upon the cardinality of $x(S)$, and *vice versa*; hence,

$$\mathbb{P}\left[d_{ij} = 1 \mid j \in x(S) \cap P(t)\right] = \mathbb{P}\left[d_{ij} = 1 \mid j \in x(S)\right]$$

and applying once again the law of total probability

$$\begin{aligned}
(**) &= \mathbb{P}\left[d_{ij} = 1 \mid j \in x(S)\right] \\
&= \sum_{s=1}^{|S|} \mathbb{P}\left[d_{ij} = 1 \mid P(s, j) \cap j \in x(S)\right] \cdot \mathbb{P}\left[P(s, j) \mid j \in x(S)\right]
\end{aligned}$$

where the summation starts from one since $j \in x(S)$ implies that $d_{ij} = 1$ for at least a cell $c_i \in S$. We can now repeat the previous reasoning: if we know that $P(s, j)$ holds for a certain $s > 0$, then $j \in x(S)$ does not impose any further constraint; thus,

$$\mathbb{P}\left[d_{ij} = 1 \mid P(s,j) \cap j \in x(S)\right] = \mathbb{P}\left[d_{ij} = 1 \mid P(s,j)\right]$$

which allows us to derive

$$\begin{aligned}(**) &= \sum_{s=1}^{|S|} \underbrace{\mathbb{P}\left[d_{ij} = 1 \mid P(s,j)\right]} \cdot \underbrace{\mathbb{P}\left[P(s,j) \mid j \in x(S)\right]} \\ &= \sum_{s=1}^{|S|} \underbrace{\frac{s}{|S|}} \cdot \underbrace{\binom{|S|}{s} \frac{(1-p)^s p^{|S|-s}}{1-p^{|S|}}}\end{aligned}$$

where the left hand side factor is derived by noticing that we have a uniform distribution. As for the right hand side factor, it is just a binomial distribution, except that a normalizing factor $1 - p^{|S|}$ is added in order to account for the fact that $j \in x(S)$, i. e. that we must exclude the case $s = 0$ when the considered mutation is not present in any cell. Cleaning up,

$$\begin{aligned}(**) &= \sum_{s=1}^{|S|} \frac{s}{|S|} \cdot \binom{|S|}{s} \frac{(1-p)^s p^{|S|-s}}{1-p^{|S|}} \\ &= \frac{1}{|S|(1-p^{|S|})} \sum_{s=1}^{|S|} s \binom{|S|}{s} (1-p)^s p^{|S|-s} && \text{linearity} \\ &= \frac{1}{|S|(1-p^{|S|})} \sum_{s=0}^{|S|} s \binom{|S|}{s} (1-p)^s p^{|S|-s} && \text{additive identity} \\ &= \frac{\mathbb{E}\left[\mathcal{B}\left(|S|, (1-p)\right)\right]}{|S|(1-p^{|S|})} && \mathbb{E} \text{ of binomial variable} \\ &= \frac{|S|(1-p)}{|S|(1-p^{|S|})} && \mathbb{E} \text{ of binomial variable} \\ &= \frac{1-p}{1-p^{|S|}}.\end{aligned}$$

Finally, we can consider $(***)$; since each mutation from $x^*(S)$ is equally likely to be included in $x(S)$, we can easily derive that

$$\mathbb{P}\left[j \in x(S) \mid P(t)\right] = \frac{t}{|x^*(S)|}.$$

The difficult part of the proof is now completed, and we can simply start to reconstruct our original conditional expectation; first of all, by substituting the newly found values of $(**)$ and $(***)$ into $(*)$, we get

$$\mathbb{E}\left[d_{ij} \mid P(t)\right] = \frac{1-p}{1-p^{|S|}} \cdot \frac{t}{|x^*(S)|}$$

and by finally substituting the value of $(*)$ into our original equation

$$\begin{aligned}
\mathbb{E}[\delta(x(S), S) \mid P(t)] &= \frac{1}{t \cdot |S|} \sum_{c_i \in S} \sum_{j \in x^*(S)} \mathbb{E}\left[d_{ij} \mid P(t)\right] \\
&= \frac{1}{t \cdot |S|} \sum_{c_i \in S} \sum_{j \in x^*(S)} \frac{1-p}{1-p^{|S|}} \cdot \frac{t}{|x^*(S)|} \\
&= \frac{1-p}{1-p^{|S|}} \cdot \frac{t}{|x^*(S)|} \cdot \frac{1}{t \cdot |S|} \sum_{c_i \in S} \sum_{j \in x^*(S)} 1 \\
&= \frac{1-p}{1-p^{|S|}} \cdot \frac{|S| \cdot |x^*(S)|}{|x^*(S)| \cdot |S|} \\
&= \frac{1-p}{1-p^{|S|}}.
\end{aligned}$$

□

Proposition 4.1.2. *Let D be a mutation matrix affected by destructive noise with strength p. Let $S \subseteq C$ be a non-empty subset of a clone originally present in the data such that $|x(S)| > 0$. Then,*

$$\mathbb{E}[\delta(x(S), S)] = \frac{1-p}{1-p^{|S|}}.$$

Proof. Applying the law of total expectation and Lemma 4.1.1, we easily get

$$\begin{aligned}
\mathbb{E}[\delta(x(S), S)] &= \sum_{t=1}^{|x^*(S)|} \mathbb{E}[\delta(x(S), S) \mid |x(S)| = t] \cdot \mathbb{P}[|x(S)| = t] \\
&= \sum_{t=1}^{|x^*(S)|} \frac{1-p}{1-p^{|S|}} \cdot \mathbb{P}[|x(S)| = t] \\
&= \frac{1-p}{1-p^{|S|}} \sum_{t=1}^{|x^*(S)|} \mathbb{P}[|x(S)| = t] \\
&= \frac{1-p}{1-p^{|S|}} \cdot 1 \\
&= \frac{1-p}{1-p^{|S|}}.
\end{aligned}$$

□

Corollary 4.1.2.1. *Let D and S be defined as per Proposition 4.1.2. Then, as $|S| \to \infty$, the expected value of $\delta(x(S), S)$ tends to $1-p$.*

Proof. Recall that by definition $p < 1$; hence, we can trivially verify that

$$\lim_{|S|\to\infty} \frac{1-p}{1-p^{|S|}} = 1-p.$$

$\square$

Remark. *Interestingly, we can verify that*

$$\lim_{p\to 1} \frac{1-p}{1-p^{|S|}} = \lim_{p\to 1} \frac{1-p}{(1-p)\sum_{t=0}^{|S|-1} p^t} = \frac{1}{\sum_{t=0}^{|S|-1} 1} = \frac{1}{|S|}$$

which, consistently with Corollary 4.1.2.1, tends to $1-p=0$ as $|S| \to \infty$.

Lemma 4.1.3. *Let $f(\alpha) = \dfrac{\alpha}{1-p^\alpha}$ for $p \in [0,1)$ and $\alpha \in \mathbb{R} > 0$. Then, $f(\alpha)$ is strictly monotonically increasing.*

Proof. We can easily verify that

$$\frac{d}{d\alpha} f(\alpha) = \frac{1-p^\alpha + p^\alpha \cdot \alpha \ln p}{(1-p^\alpha)^2} > 0 \quad \Leftrightarrow \quad \underbrace{1-p^\alpha + p^\alpha \cdot \alpha \ln p}_{(*)} > 0.$$

We can rearrange $(*)$ as follows:

$$\begin{aligned}(*) &= 1 - e^{\alpha \ln p} + e^{\alpha \ln p} \cdot \alpha \ln p \\ &= 1 - e^t + te^t\end{aligned}$$

where the substitution $t = \alpha \ln p$ was applied. We can easily verify that for $t = 0$ we get $1 - e^t + te^t = 0$; otherwise, we can differentiate once again, getting

$$\frac{d}{dt}\left(1 - e^t + te^t\right) = 0 - e^t + e^t + te^t = te^t$$

which is strictly positive for $t > 0$ and strictly negative for $t < 0$. Hence, $1 - e^t + te^t > 0 \; \forall t \neq 0$. But this implies that

$$(*) > 0 \quad \Leftrightarrow \quad \alpha \ln p \neq 0$$

which always holds since $\alpha > 0$ and $p \neq 1$. $\square$

Proposition 4.1.4. *Let D be a mutation matrix affected by destructive noise with strength p. Let $S \subseteq C$ be a non-empty set of cells that originally belonged to at least two different clones. Then,*

$$\mathbb{E}[\delta(x(S), S)] < \frac{1-p}{1-p^{|S|}}.$$

Proof. The informal idea works as follows: first, we will divide all the mutations that could possibly be included into disjoint groups; then, for each group of mutations, we will consider all the cells that could include all the given mutations. Each such pair of cells and mutations will then behave as if it were a clone, enabling us to exploit the previous results. But since we are considering at least two clones there is always a group of mutations that cannot be included by all cells. Therefore, we are always going to have some non-zero probability of including such mutations, which will decrease the average density by forcing us to include some "extra" zeros in our computation. We will then be able to conclude by properly weighting and summing the contribution of each group of mutations.

Before starting the actual proof, we need to introduce some notation. Let us partition S into disjoint subsets $S_1, S_2, \ldots, S_T$ such that each S_t contains all and only the cells from S that originally belonged to the same clone. It is trivial to check that $\bigcup_{t=1}^{T} S_t = S$.

Let then $\mathcal{S} = \{S_1, S_2, \ldots, S_T\}$, and let $x^*(\mathbf{s}) \subset x^*(S)$ be the set of mutations that originally were exclusive to the set of clones $\mathbf{s} \subseteq \mathcal{S}$. Notice that it may happen that $x^*(\mathbf{s}) = \emptyset$, and that by construction all the $x^*(\mathbf{s})$ create a partitioning of $x^*(S)$, since each mutation is included in exactly one such set. Similarly, let $x(\mathbf{s}) = x^*(\mathbf{s}) \cap x(S)$, i.e. the set of mutations that originally were exclusive to the set of clones $\mathbf{s} \in \mathcal{S}$, and that are actually present in $x(S)$. Finally, let

$$\mathcal{P} = \{\mathbf{s} \subseteq \mathcal{S} \text{ s.t. } x^*(\mathbf{s}) \neq \emptyset\}$$

and

$$S(\mathbf{s}) = \bigcup_{S_t \in \mathbf{s}} S_t.$$

Notice that $\{x(\mathbf{s}) \text{ s.t. } \mathbf{s} \in \mathcal{P}\}$ is still a partitioning of $x(S)$, since we only excluded empty sets (which do not contribute to the resulting union).

The notation we just introduced is relatively heavy, so it is quite helpful to fix the idea by looking at a simple example, for which we can once again refer to Figure 2.1. In that case, we would have:

- $\mathcal{S} = \{S_1 \subseteq C_1, S_2 \subseteq C_2\}$
- $x^*(\mathbf{s}) = V_2$ for $\mathbf{s} = \{S_2\}$
- $x^*(\mathbf{s}) = V_1$ for $\mathbf{s} = \{S_1, S_2\}$

- $x^*(\mathbf{s}) = \emptyset$ for $\mathbf{s} = \{S_1\}$
- $\mathcal{P} = \{\{S_2\}, \{S_1, S_2\}\}$

We can now start the actual proof. First of all, we can verify that for each $\mathbf{s} \in \mathcal{P}$ it holds true that

$$\sum_{c_i \in S} \sum_{j \in x(\mathbf{s})} d_{ij} = \sum_{c_i \in S(\mathbf{s})} \sum_{j \in x(\mathbf{s})} d_{ij} \tag{4.2}$$

since by construction $\nexists d_{ij} = 1$ s.t. $j \in x(\mathbf{s}) \wedge c_i \notin S(\mathbf{s})$. Therefore,

$$\begin{aligned}
\delta(x(S), S) &= \frac{\sum_{c_i \in S} \sum_{j \in x(S)} d_{ij}}{|x(S)| \cdot |S|} && \text{by definition} \\
&= \frac{\sum_{c_i \in S} \sum_{\mathbf{s} \in \mathcal{P}} \sum_{j \in x(\mathbf{s})} d_{ij}}{|x(S)| \cdot |S|} && \text{by linearity} \\
&= \sum_{\mathbf{s} \in \mathcal{P}} \frac{\sum_{c_i \in S} \sum_{j \in x(\mathbf{s})} d_{ij}}{|x(S)| \cdot |S|} && \text{by linearity} \\
&= \sum_{\mathbf{s} \in \mathcal{P}} \underbrace{\frac{\sum_{c_i \in S(\mathbf{s})} \sum_{j \in x(\mathbf{s})} d_{ij}}{|x(S)| \cdot |S|}}_{(*)} && 4.2
\end{aligned}$$

Let $X(\mathbf{s})$ be the random variable with value

$$X(\mathbf{s}) = \begin{cases} \frac{|x(\mathbf{s})|}{|x(S)|} \cdot \frac{|S(\mathbf{s})|}{|S|} \cdot \delta(x(\mathbf{s}), S(\mathbf{s})) & \text{if } |x(\mathbf{s})| \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

and notice that

$$\begin{aligned}
(*) &= 0 = X(\mathbf{s}) && \text{if } |x(\mathbf{s})| = 0 \\
(*) &= \frac{|x(\mathbf{s})| \cdot |S(\mathbf{s})|}{|x(S)| \cdot |S|} \cdot \frac{\sum_{c_i \in S(\mathbf{s})} \sum_{j \in x(\mathbf{s})} d_{ij}}{|x(\mathbf{s})| \cdot |S(\mathbf{s})|} = X(\mathbf{s}) && \text{otherwise}
\end{aligned}$$

where in the first case we exploited the fact that the numerator contains a sum over the elements of an empty set, and the second case is a simple algebraic manipulation followed by the application of the definition of density. Therefore,

$$\delta(x(S), S) = \sum_{\mathbf{s} \in \mathcal{P}} X(\mathbf{s}).$$

Let $k = |\mathcal{P}|$, and let $\mathbf{a} = [|x(\mathbf{s_1})|, |x(\mathbf{s_2})|, \ldots, |x(\mathbf{s_k})|]^T$; recall that each entry of $\mathbf{a}$ is a random variable. Let $\mathbf{a_s}$ be the entry of $\mathbf{a}$ corresponding to $x(\mathbf{s})$. Finally, let A be the set of admissible values for $\mathbf{a}$. Notice that thanks to the way we constructed $\mathcal{P}$ there is always at least one $\mathbf{a}' \in A$ such that $\mathbf{a}'_{\mathbf{s}} > 0$, for each $\mathbf{s} \in \mathcal{P}$. Then,

$$\begin{aligned}
\mathbb{E}\left[\delta(x(S), S)\right] &= \mathbb{E}\left[\sum_{\mathbf{s}\in\mathcal{P}} X(\mathbf{s})\right] \\
&= \sum_{\mathbf{s}\in\mathcal{P}} \mathbb{E}\left[X(\mathbf{s})\right] && \text{linearity of } \mathbb{E} \\
&= \sum_{\mathbf{s}\in\mathcal{P}} \sum_{\mathbf{a}'\in A} \mathbb{E}\left[X(\mathbf{s}) \mid \mathbf{a} = \mathbf{a}'\right] \cdot \mathbb{P}\left[\mathbf{a} = \mathbf{a}'\right]
\end{aligned}$$

where in the last step we applied the law of total expectation. Next, we can verify that as long as $\mathbf{a}'_{\mathbf{s}} \neq 0$

$$\begin{aligned}
\mathbb{E}\left[X(\mathbf{s}) \mid \mathbf{a} = \mathbf{a}'\right] &= \mathbb{E}\left[\frac{|x(\mathbf{s})|}{|x(S)|} \cdot \frac{|S(\mathbf{s})|}{|S|} \cdot \delta(x(\mathbf{s}), S(\mathbf{s})) \;\middle|\; \mathbf{a} = \mathbf{a}'\right] \\
&= \mathbb{E}\left[\frac{\mathbf{a}'_{\mathbf{s}}}{\sum_{\mathbf{s}'\in\mathcal{P}} \mathbf{a}'_{\mathbf{s}'}} \cdot \frac{|S(\mathbf{s})|}{|S|} \cdot \delta(x(\mathbf{s}), S(\mathbf{s})) \;\middle|\; \mathbf{a} = \mathbf{a}'\right] \\
&= \frac{\mathbf{a}'_{\mathbf{s}}}{\sum_{\mathbf{s}'\in\mathcal{P}} \mathbf{a}'_{\mathbf{s}'}} \cdot \frac{|S(\mathbf{s})|}{|S|} \cdot \mathbb{E}\left[\delta(x(\mathbf{s}), S(\mathbf{s})) \mid \mathbf{a} = \mathbf{a}'\right]
\end{aligned}$$

where the last step is due to the linearity of the conditional expected value. We can now notice that $\delta(x(\mathbf{s}), S(\mathbf{s}))$ is independent from all elements of $\mathbf{a}$ except $\mathbf{a}_{\mathbf{s}}$; therefore,

$$\mathbb{E}\left[\delta(x(\mathbf{s}), S(\mathbf{s})) \mid \mathbf{a} = \mathbf{a}'\right] = \mathbb{E}\left[\delta(x(\mathbf{s}), S(\mathbf{s})) \mid \mathbf{a}_{\mathbf{s}} = \mathbf{a}'_{\mathbf{s}}\right].$$

Furthermore, $x(\mathbf{s})$ and $S(\mathbf{s})$ are structured like a clone with its corresponding mutations; hence, we can apply Lemma 4.1.1, yielding

$$\begin{aligned}
\mathbb{E}\left[\delta(x(\mathbf{s}), S(\mathbf{s})) \mid \mathbf{a} = \mathbf{a}'\right] &= \mathbb{E}\left[\delta(x(\mathbf{s}), S(\mathbf{s})) \mid \mathbf{a}_{\mathbf{s}} = \mathbf{a}'_{\mathbf{s}}\right] \\
&= \mathbb{E}\left[\delta(x(\mathbf{s}), S(\mathbf{s})) \mid |x(\mathbf{s})| = \mathbf{a}'_{\mathbf{s}}\right] && \text{by definition of } \mathbf{a} \\
&= \frac{1-p}{1-p^{|S(\mathbf{s})|}} && \text{Lemma 4.1.1}
\end{aligned}$$

Let us now consider the inequality

$$\begin{aligned}
\frac{|S(\mathbf{s})|}{|S|} \cdot \mathbb{E}\left[\delta(x(\mathbf{s}), S(\mathbf{s})) \mid \mathbf{a} = \mathbf{a}'\right] &< \frac{1-p}{1-p^{|S|}} \\
\frac{|S(\mathbf{s})|}{|S|} \cdot \frac{1-p}{1-p^{|S(\mathbf{s})|}} &< \frac{1-p}{1-p^{|S|}} \\
\frac{|S(\mathbf{s})|}{|S|} \cdot \frac{1}{1-p^{|S(\mathbf{s})|}} &< \frac{1}{1-p^{|S|}} && p \neq 1 \\
\frac{|S(\mathbf{s})|}{1-p^{|S(\mathbf{s})|}} &< \frac{|S|}{1-p^{|S|}} && |S| > 0
\end{aligned}$$

and thanks to Lemma 4.1.3 we can conclude that the considered inequality holds as long as $|S| > |S(\mathbf{s})|$. This is indeed the case for all $\mathbf{s} \in \mathcal{P}$ except at most one where $S(\mathbf{s}) = S$. Notice that such element may not be present (i. e., there might not be any mutation that originally was present in all the considered cells). On the other hand, there must for sure be at least one element where $|S| > |S(\mathbf{s})|$, since we are considering cells from at least two different clones and therefore there must be at least a mutation that originally was not present in all the considered cells. All in all, this means that for each $\mathbf{s} \in \mathcal{P}$

$$\begin{aligned}\mathbb{E}\left[X(\mathbf{s}) \mid \mathbf{a} = \mathbf{a}'\right] &= \frac{\mathbf{a}'_{\mathbf{s}}}{\sum_{\mathbf{s}' \in \mathcal{P}} \mathbf{a}'_{\mathbf{s}'}} \cdot \frac{|S(\mathbf{s})|}{|S|} \cdot \mathbb{E}\left[\delta(x(\mathbf{s}), S(\mathbf{s})) \mid \mathbf{a} = \mathbf{a}'\right] \\ &< \frac{\mathbf{a}'_{\mathbf{s}}}{\sum_{\mathbf{s}' \in \mathcal{P}} \mathbf{a}'_{\mathbf{s}'}} \cdot \frac{1-p}{1-p^{|S|}}\end{aligned}$$

except at most one case where we have an equality. We did not consider the case $\mathbf{a}'_{\mathbf{s}} = 0$ yet; however, we can trivially verify that we still have an equality, since in this case

$$\begin{aligned}\mathbb{E}\left[\delta(x(S), S)\right] &= 0 \\ &= \frac{\mathbf{a}'_{\mathbf{s}}}{\sum_{\mathbf{s}' \in \mathcal{P}} \mathbf{a}'_{\mathbf{s}'}} \cdot \frac{1-p}{1-p^{|S|}}.\end{aligned}$$

This implies that

$$\begin{aligned}\mathbb{E}\left[\delta(x(S), S)\right] &= \sum_{\mathbf{s} \in \mathcal{P}} \sum_{\mathbf{a}' \in A} \mathbb{E}\left[X(\mathbf{s}) \mid \mathbf{a} = \mathbf{a}'\right] \cdot \mathbb{P}\left[\mathbf{a} = \mathbf{a}'\right] \\ &< \sum_{\mathbf{s} \in \mathcal{P}} \sum_{\mathbf{a}' \in A} \frac{\mathbf{a}'_{\mathbf{s}}}{\sum_{\mathbf{s}' \in \mathcal{P}} \mathbf{a}'_{\mathbf{s}'}} \cdot \frac{1-p}{1-p^{|S|}} \cdot \mathbb{P}\left[\mathbf{a} = \mathbf{a}'\right] \\ &= \frac{1-p}{1-p^{|S|}} \cdot \sum_{\mathbf{s} \in \mathcal{P}} \sum_{\mathbf{a}' \in A} \frac{\mathbf{a}'_{\mathbf{s}}}{\sum_{\mathbf{s}' \in \mathcal{P}} \mathbf{a}'_{\mathbf{s}'}} \cdot \mathbb{P}\left[\mathbf{a} = \mathbf{a}'\right] && \text{by linearity} \\ &= \frac{1-p}{1-p^{|S|}} \cdot \sum_{\mathbf{a}' \in A} \mathbb{P}\left[\mathbf{a} = \mathbf{a}'\right] \cdot \sum_{\mathbf{s} \in \mathcal{P}} \frac{\mathbf{a}'_{\mathbf{s}}}{\sum_{\mathbf{s}' \in \mathcal{P}} \mathbf{a}'_{\mathbf{s}'}} && \text{by linearity} \\ &= \frac{1-p}{1-p^{|S|}} \cdot \sum_{\mathbf{a}' \in A} \mathbb{P}\left[\mathbf{a} = \mathbf{a}'\right] \cdot \frac{\sum_{\mathbf{s} \in \mathcal{P}} \mathbf{a}'_{\mathbf{s}}}{\sum_{\mathbf{s}' \in \mathcal{P}} \mathbf{a}'_{\mathbf{s}'}} && \text{by linearity} \\ &= \frac{1-p}{1-p^{|S|}} \cdot \sum_{\mathbf{a}' \in A} \mathbb{P}\left[\mathbf{a} = \mathbf{a}'\right] \cdot 1 \\ &= \frac{1-p}{1-p^{|S|}} \cdot 1 \\ &= \frac{1-p}{1-p^{|S|}}.\end{aligned}$$

$\square$

Corollary 4.1.4.1. *Let D and S be defined as per Proposition 4.1.4. Then,*

$$\lim_{|S|\to\infty} \mathbb{E}[\delta(x(S), S)] \leq 1 - p.$$

Proof. Applying Proposition 4.1.4 and the sandwich theorem, we easily get

$$\begin{aligned}\lim_{|S|\to\infty} \mathbb{E}[\delta(x(S), S)] &\leq \lim_{|S|\to\infty} \frac{1-p}{1-p^{|S|}} \\ &= 1 - p.\end{aligned}$$

□

All in all, what we have just proved highlights the presence of a deep connection between density and the amount of noise in matrix D. Some consequences of such connection will be explored in more detail in the following Sections; at the moment, the most important conclusion is that the proposed score is provably meaningful. In particular, true clones do reach in expectation a strictly better score with respect to randomly picked sets of cells, and even asymptotically the expected score they reach is not worse. In other words, it is reasonable to use density as an heuristic to identify the correct itemsets within noisy data; as previously seen at the end of Section 2.2, this is exactly what we needed. However, this does not mean that we cannot achieve a better result, as we will see in the following Sections.

4.1.2 Weighted density

The first enhancement to the basic definition of density is empirically motivated. Let us consider once again the example of Figure 2.1, and this time suppose that $|V_1| \gg |V_2|$. Intuitively, this means that it's harder to distinguish cells from C_1 and cells of C_2, since cells will contain mostly the same mutations anyway. Notice how this problem is linked to the issue with overlapping itemsets mentioned for the ETI algorithm in Section 3.1. We can then introduce the following quantity:

Definition 4.1.2 (weight of a mutation)**.** Let D be affected by destructive noise. Let $j \in V$ be a mutation, and suppose that $\exists c_i \in C$ s.t. $d_{ij} = 1$. Then, its *weight* is defined as

$$w_j = \frac{1}{\sum_{c_i \in C} d_{ij}}.$$

Remark. *Notice that in practice if $\nexists c_i \in C$ s.t. $d_{ij} = 1$ the mutation should be dropped from our matrix, since we have no way of telling which cells may or may not contain such mutation.*

Such weights naturally lead to the following definition:

Definition 4.1.3 (Weighted density)**.** Let $x \subseteq V$ be a generic non-empty itemset. Let $S \subseteq C$ be a non-empty set of transactions. Then, the *weighted density of* x *in* S is defined as

$$\delta_w(x, S) = \frac{\sum_{c_i \in S} \sum_{j \in x} w_j d_{ij}}{\left(\sum_{j \in x} w_j\right) \cdot |S|}.$$

As usual in practice we will take $x = x(S)$. Finding exact expected values for w_j and $\delta_w(x, S)$ is not trivial due to the bias introduced by requiring that the mutation is present at least once; however, it seems reasonable to expect an higher weight from mutations that before the introduction of noise were supported by fewer cells. In our example, this would correspond to giving more importance to mutations from V_2; in turns, this should intuitively help distinguishing very similar clones.

We will see in Chapter 5, and in particular in Section 5.1, that in practice weighted density does indeed perform better than plain density; but there is still a way to further improve our score.

4.1.3 Adjusted density

An important consequence of Proposition 4.1.2 is that the expected value of density is strictly decreasing as the number of cells considered increases. This means that on average smaller groups of cells will be preferred even over the true original clones, which of course is not a desirable behaviour. The natural solution is to introduce the following.

Definition 4.1.4 (Adjusted density)**.** Let $S \subseteq C$ be a non-empty set of transactions. Let $x \subseteq V$ be a generic non-empty itemset. Let $I(S) = \{S' \subseteq C \text{ s.t. } |S'| = |S|\}$ be the set of all subsets of C with the same cardinality as S. Then, the *adjusted density of* x *in* S is defined as

$$\delta_A(x, S) = \delta(x, S) \; - \underset{S' \sim \mathcal{U}(I(S))}{\mathbb{E}} \left[\delta(x(S'), S')\right]$$

where $\mathcal{U}\left(I(S)\right)$ is the uniform distribution over $I(S)$.

Similarly, we can define the following.

Definition 4.1.5 (Adjusted weighted density)**.** Let S, x, $I(S)$ and $\mathcal{U}\left(I(S)\right)$ be as in Definition 4.1.4. Then, the *adjusted weighted density of* x *in* S is defined as

$$\delta_{A,w}(x, S) = \delta_w(x, S) \; - \underset{S' \sim \mathcal{U}(I(S))}{\mathbb{E}} \left[\delta_w(x(S'), S')\right].$$

Notice that in both cases computing the value of the score is not trivial, due to the presence of an expected value. We have a closed-form expression for $\mathbb{E}[\delta(x(S'), S')]$ when S' is a clone, and a bound otherwise; but even this is not extremely helpful in practice: the problem is that in general the value of p can only be estimated for the sequencing technology of choice, and is thus not known with precision.

The proposed solution is to use sampling so as to estimate the expected value of interest directly from data; such approach will be presented in more detail in Section 4.2.3, while some considerations and tests about the number of samples needed to get a good approximation can be found in Sections 4.4 and 5.6 respectively.

4.2 Algorithms to optimize density

In the previous Section we have seen several scores that can be used to choose between different itemsets; the second task we have to consider is how to select the itemsets that maximize the score of choice. The simplest option would be to just enumerate all possible itemsets and choose the best performing ones; however, the number of different options to consider is exponential in the number of mutations, making this approach clearly unfeasible. An interesting tradeoff between output quality and time complexity can instead be reached by using a greedy agglomerative clustering algorithm, as we will see next.

4.2.1 Naive algorithm

Let us start from the simplest option, where we simply try to blindly merge together subsets of cells that maximize density; the result is Algorithm 1.

In practice, this algorithm is far from the best we can do; however, it is an ideal foundation upon which we can incrementally build more advanced algorithms. Furthermore, it is a good test bench to experimentally define a performance baseline, and to evaluate how well density alone works in practice.

Anyway, the algorithm works as follows. First of all, we denote with $\mathcal{C}_{index}$ a clustering of all the considered cells, where along with each cluster $S \in \mathcal{C}_{index}$ we also store the corresponding itemset $x(S)$. The first clustering (line 3) is initially filled (lines 4 to 6) by inserting each cell in its own cluster (line 5). We can then start to iteratively merge clusters together.

```
Algorithm 1: CLUSTER_NAIVE
  Input: Mutation matrix $D$, list of cells $C$, list of mutations $V$.
  Output: Dendrogram of cell clusters and corresponding itemsets of
          mutations.
1 begin
2     $index \leftarrow 1$;
3     $\mathcal{C}_1 \leftarrow$ empty set;
4     foreach cell $c_i \in C$ do
5         $S_i \leftarrow \{c_i\}$;
6         $\mathcal{C}_1 \leftarrow \mathcal{C}_1 \cup \{\, (S_i,\ x(S_i)) \,\}$;
7     while not all cells are in a single cluster do
8         $(S_a,\ x(S_a)),\ (S_b,\ x(S_b)) \leftarrow$ elements of $\mathcal{C}_{index}$ maximizing
            $\delta(x(S_a \cup S_b),\ S_a \cup S_b)$;
9         $\mathcal{C}_{index+1} \leftarrow \mathcal{C}_{index} \setminus \{\, (S_a,\ x(S_a)),\ (S_b,\ x(S_b)) \,\}$;
10        $\mathcal{C}_{index+1} \leftarrow \mathcal{C}_{index+1} \cup \{\, (S_a \cup S_b,\ x(S_a \cup S_b)) \,\}$;
11        $index \leftarrow index + 1$;
12    return $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m$;
```

The clusters to be merged are simply the ones that maximize the density of the resulting cluster (line 8). Once we have identified such pair, to obtain the new clustering we just remove the old clusters (line 10) and replace them with the merged one (line 10). After increasing the index of the iteration (line 11) we can simply repeat the process, until we are left with a single cluster containing all cells (line 7). Finally, we can output our result (line 12) which is composed of all the clustering we obtained at the end of each iteration; since each time we merge two clusters we reduce by one the cardinality of the resulting clustering, and since the cardinality of the first clustering is equal to the cardinality m of the set C of cells, it is immediate to verify that we will output exactly m different clusterings.

According to our problem definition, we now need to choose a single clustering among the ones that the algorithm returned; to do so, we need some kind of model selection. Only then we will be able to separate the individual groups of mutation, as described in Section 2.2. However, there are already several model selection algorithms available in literature; furthermore, in real-world scenarios we may actually be interested in analyzing the whole dendrogram, either manually or automatically. Therefore, it is a good idea to keep model selection separate from the clustering algorithm.

4.2.2 Weighted algorithm

The second proposed algorithm looks very similar to the first one, but two important changes are made. First of all, we will use weighted density in place of just density; secondly, we will try whenever possible to merge only those groups of cells that share at least a mutation in the noisy matrix.

It is quite obvious that merging clusters that do not share any mutation does not make much sense, since there is no proof that such cells originally did not belong to different clones. However, this kind of union can happen relatively frequently in practice: for example, if we let S be composed of two cells at random that do not share any mutation, it is trivial to verify that according to definition $\delta(x(S), S) = 0.5$.

If we then take $p = 0.995$, a common value through the tests presented in [2], we get that the expected value of the density from two cells of the same clone is just $\frac{1-0.995}{1-0.995^2} \approx 0.5013$. Quite clearly, the difference is low enough to allow for several errors, at least in our greedy setup. If we instead consider weighted density things get more complex; however, the underlying idea is still valid, hence the need to avoid merging non-intersecting groups of mutations as much as possible. All in all, this results in Algorithm 2.

This algorithm is very similar to the first one, with the only differences being in lines 8 to 11. Here, we are doing exactly what was described at the start of this Section: first, we check if we can merge clusters whose cells share at least a mutation (line 8); if this is the case, we simply pick the pair of clusters with at least a common mutation that maximize weighted density when merged (line 9). If no such pair can be found, we simply pick the one with the best resulting score (line 11).

In practice, despite the apparent similarity, Algorithm 2 consistently outperforms Algorithm 1, as we will see in Chapter 5; however, there is still one last option to consider.

Algorithm 2: CLUSTER_WEIGHTED

```
Input: Mutation matrix $D$, list of cells $C$, list of mutations $V$.
Output: Dendrogram of clones and corresponding itemsets of mutations.
1  begin
2      $index \leftarrow 1$;
3      $\mathcal{C}_1 \leftarrow$ empty set;
4      foreach cell $c_i \in C$ do
5          $S_i \leftarrow \{c_i\}$;
6          $\mathcal{C}_1 \leftarrow \mathcal{C}_1 \cup \{\,(S_i,\ x(S_i))\,\}$;
7      while not all cells are in a single cluster do
8          if $\exists\,(S_a,\ x(S_a)),\ (S_b,\ x(S_b)) \in \mathcal{C}_{index}$ s.t. $x(S_a) \cap x(S_b) \neq \emptyset$ then
9              $(S_a,\ x(S_a)),\ (S_b,\ x(S_b)) \leftarrow$ elements of $\mathcal{C}_{index}$ such that
               $x(S_a) \cap x(S_b) \neq \emptyset$ and $\delta_w(x(S_a \cup S_b),\ S_a \cup S_b)$ is
               maximized;
10         else
11             $(S_a,\ x(S_a)),\ (S_b,\ x(S_b)) \leftarrow$ elements of $\mathcal{C}_{index}$ such that
               $\delta_w(x(S_a \cup S_b),\ S_a \cup S_b)$ is maximized;
12         $\mathcal{C}_{index+1} \leftarrow \mathcal{C}_{index} \setminus \{\,(S_a,\ x(S_a)),\ (S_b,\ x(S_b))\,\}$;
13         $\mathcal{C}_{index+1} \leftarrow \mathcal{C}_{index+1} \cup \{\,(S_a \cup S_b,\ x(S_a \cup S_b))\,\}$;
14         $index \leftarrow index + 1$;
15     return $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m$;
```

4.2.3 Sampling algorithm

We have previously introduced three different variations of the same score, two of which where then used to build an algorithm. It is all but natural to exploit the last variation, too. The general idea is to reuse the structure of Algorithm 2 and just replace δ_w with $\delta_{A,w}$; however, we need to add a new subroutine to approximate the required expected values. All in all, this results in Algorithm 3.

Lines 8 and 10 just reflect the change in score we just described, while most other lines are just unchanged; as such, we can focus on the auxiliary procedure starting at line 15. To keep a good amount of efficiency, it is quite clear that we cannot recompute the expected values each time we need them. We may then try to precompute them all; however, the total number of expected values we may theoretically need is linear in the cardinality of C, while the number of expected values we will use in practice is experimentally only a small subset of the total ones. Therefore, the best option is to rely on memoization: we first initialize an empty dictionary (line 5) that we will fill as the algorithm runs; then, each time an expected value is required, we check if we already computed its approximation (line 16). If this is the case, we do not enter the body of the if clause and we simply return the memoized value from the dictionary (line 22). If instead the expected value was never required before, then we initialize a new variable to contain our approximation (line 17). Such approximation will be obtained by sampling the required number of cells uniformly at random from the entire population (line 19) and by updating the average weighted density with the considered sample (line 20). The whole process is repeated for a fixed number of iterations (line 18); finally, the approximated expected value is inserted into out dictionary (line 21) and we can conclude by returning the requested expectation (line 22).

This procedure is by no means complex; it does however raise the problem of choosing a proper amount of samples to estimate the expected value with a reasonable amount of precision. As previously mentioned, this will be the topic of Sections 4.4 and 5.6.

Once again, we will see in Chapter 5 how this slight variation of the general greedy clustering approach affects the overall performance of the algorithm; however, we first need to consider some implementation details that up to now we hid for simplicity.

Algorithm 3: CLUSTER_ADJUSTED

```
Input: Mutation matrix $D$, list of cells $C$, list of mutations $V$.
Output: Dendrogram of clones and corresponding itemsets of mutations.
1  begin
2      $index \leftarrow 1$;   $\mathcal{C}_1 \leftarrow$ empty set;
3      foreach $cell\ c_i \in C$ do
4          $S_i \leftarrow \{c_i\}$;   $\mathcal{C}_1 \leftarrow \mathcal{C}_1 \cup \{\,(S_i,\ x(S_i))\,\}$;
5      $memo \leftarrow$ empty dictionary;
6      while not all cells are in a single cluster do
7          if $\exists\,(S_a,\ x(S_a)),\ (S_b,\ x(S_b)) \in \mathcal{C}_{index}$ s.t. $x(S_a) \cap x(S_b) \neq \emptyset$ then
8              $(S_a,\ x(S_a)),\ (S_b,\ x(S_b)) \leftarrow$ elements of $\mathcal{C}_{index}$ such that
                 $x(S_a) \cap x(S_b) \neq \emptyset$ and $\delta_w(x(S_a \cup S_b),\ S_a \cup S_b) -$
                 getExpVal($memo,\ |S_a \cup S_b|,\ C,\ V,\ D$) is maximized;
9          else
10             $(S_a,\ x(S_a)),\ (S_b,\ x(S_b)) \leftarrow$ elements of $\mathcal{C}_{index}$ such that
                 $\delta_w(x(S_a \cup S_b),\ S_a \cup S_b) -$ getExpVal($memo,\ |S_a \cup S_b|,$
                 $C,\ V,\ D$) is maximized;
11         $\mathcal{C}_{index+1} \leftarrow \mathcal{C}_{index} \setminus \{\,(S_a,\ x(S_a)),\ (S_b,\ x(S_b))\,\}$;
12         $\mathcal{C}_{index+1} \leftarrow \mathcal{C}_{index+1} \cup \{\,(S_a \cup S_b,\ x(S_a \cup S_b))\,\}$;
13         $index \leftarrow index + 1$;
14     return $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m$;
15 Procedure getExpVal($memoized,\ k,\ C,\ V,\ D$)
16     if $k \notin memoized.keys()$ then
17         $avg \leftarrow 0$;
18         for $i \leftarrow 1$ to $SAMPLES\_COUNT$ do
19             $S \leftarrow k$ cells sampled uniformly at random from $C$;
20             $avg \leftarrow avg + \frac{\delta_w(x(S), S)}{SAMPLES\_COUNT}$;
21         $memoized[k] \leftarrow avg$;
22     return $memoized[k]$;
```

4.3 Time complexity and improvements

If we want to use an algorithm in practice we do not need it to simply give good results: we need it to run in a reasonable amount of time, too. Up to now we did not consider the proposed algorithms' time complexity; while this was helpful to grasp the idea behind them, we must verify that the resulting runtimes are practical.

Let us consider the most complex option, i. e. Algorithm 3. The first cycle (line 3) is repeated m times; assuming sets are implemented as hash tables, each operation in the first cycle has on average constant complexity. The only exception is identifying $x(S_i)$ (line 4), which requires $O(n)$ operations; this means that the first cycle has complexity $O(mn)$.

As for the second cycle (line 6) it gets executed $O(m)$ times, as previously argued. The clustering $\mathcal{C}_{index}$ contains $O(m)$ elements, too; this means that the if clause of line 7 must consider $O(m^2)$ different pairs of clusters, and checking for intersection has complexity $O(n)$. Maximizing the approximation of the adjusted density, either on line 8 or on line 10, requires once again to scan $O(m^2)$ different pairs; the weighted density can be computed quite efficiently from the weighted density of each of the considered clusters: the numerator is the sum of the two numerators (available in constant time), the cardinality of the set of cells is the sum of the cardinalities of the two sets (once again, available in constant time), computing $x(S_a \cup S_b)$ can be done in $O(n)$ operations starting from $x(S_a)$ and $x(S_b)$, and finally summing the weights of the mutations in $x(S_a \cup S_b)$ can be done once again in time $O(n)$, assuming the weight of each mutation is pre-computed. To sum up, up to now we have $O(m^2 n)$ operation from the condition of the if clause, and $O(m^2 n)$ operations from either the body of the if clause or from the body of the else clause.

Each call to the auxiliary procedure to approximate the expected values requires constant time if the requested expected value was already computed. Otherwise, we can assume that $SAMPLES\_COUNT$ is a constant and can hence be neglected; this we we get complexity $O(k + kn) \in O(mn)$. We can notice that whenever the procedure has constant runtime it does not contribute to the overall complexity; since at most $O(m)$ different expected values need to be computed, the auxiliary procedure requires at most $O(m^2 n)$ operations cumulatively.

Most of the complex part of the analysis is completed: line 11 requires time $O(mn)$, while line 12 and 13 require respectively time $O(m+n)$ and $O(1)$. All in all, this means that the whole cycle requires

$$O\left(m \cdot (m^2n + m^2n + mn + m + n + 1)\right) \in O(m^3n)$$

operations.

We are almost done: all the operations outside the cycles have constant complexity, but as mentioned we need to pre-compute the weight of each mutation; this adds an additional $O(mn)$ operations to the algorithm, which as a whole requires $O(mn + m^3n + m^2n + mn)$ operations (just to recap, the four terms correspond respectively to the first cycle, the second one, the calls to the auxiliary procedure and the pre-computation of the mutations' weights). All in all, this means that the algorithm has complexity $O(m^3n)$.

This is not a terrible result; however, as the number of cells increases the algorithm can rapidly become impractical. Luckily, there is something we can do about it: the basic idea is that we are wasting a huge amount of time by repeatedly calculating the same densities over and over again, since each time that a pair of cells clusters do not get merged we have to recompute the density of their union in the next iteration.

The solution to this problem is to compute each density just once, and then store it in a max-heap along with the corresponding pair of clusters if and only if said pair contains intersecting mutation sets. Each time we need to merge two clusters we then just need to retrieve the first element of the heap and check if each of the corresponding clusters is still to be merged; if this is the case, we merge them; otherwise, we move on with the next entry of the heap. There is of course an exception to consider when no intersecting pair is available, as we will see; the result is Algorithm 4. The auxiliary procedure is exactly the same as the one of Algorithm 3 and was thus omitted.

The algorithm works as follows: we initialize our max-heap H at line 5; then, we start to fill it with all possible pairs of cells that we can merge at first (lines 6 to 10) along with the score of the resulting cluster (computed through lines 8 and 9), provided that the considered cells share at least a mutation (line 7). We can then heapify H (line 11).

We can now proceed almost as usual: we try to get the first element out of our heap (line 13); if no element is available and still not all cells are in the same cluster, it means that there is no pair of clusters that share at least a mutation left. Therefore, we just pick the pair of clusters that maximizes the resulting adjusted density, ignoring the fact that they do not share any mutation (lines 15 and 16). We then check whether one or both of the clusters obtained through the steps we just described have already been merged with another cluster (line 17). Notice that this can never be the case if we passed through the else clause of line 15, since in that case we are always picking two clusters among the current ones. Anyway, if at least one of the considered clusters has already been merged, we just proceed to the next iteration without changing the current clustering. On the other hand, if we are free to merge the considered clusters we do so as usual (lines 19 to 21) and we naturally mark them as merged (line 18).

At this point we need to update the heap with all the new possible pairs of clusters we could merge; notice how all such pairs must involve the newly created cluster. Quite trivially, this is done thanks to lines 22 to 26.

We can now focus on the resulting time complexity. Line 5 is of course executed in constant time; the cycle of line 6 gets repeated $O(m^2)$ times, with each iteration requiring $O(n)$ operations ignoring the call to the auxiliary function (whose contribute we can compute separately as previously done). This result in complexity $O(m^2n)$. Line 11 requires time linear in the size of H, i. e. $O(m^2)$.

Things now get a little more complex, since the main cycle of line 12 may be executed for a variable number of iterations. The easiest option is to separately consider the cases where we enter in the body of the if clause of line 17 (case A), and the case where we do not (case B).

First of all, notice that whenever we execute lines 18 to 27 we are reducing by one the number of clusters in the current clustering; as such, we can fall in case A no more than m times. Furthermore, let us assume for the moment that the else case of lines 15 and 16 gets executed at most $O(1)$ times; we will justify this assumption later. If this is the case, the else clause only contributes with $O(m^2n)$ operations globally, and can be ignored in the analysis of our cycle. We then have an operation of constant cost on line 13, while line 14 contributes with $O(\log(m))$. Lines 17 and 18 require once again $O(1)$, while lines 19 to 21 all in all require $O(mn)$ just as their corresponding lines from Algorithm 3. The for cycle of line 22 gets executed $O(m)$ times, with each iteration requiring $O(n)$ operations for lines 23 to 25 (since as previously argued we can compute efficiently $x(S_c \cup S_d)$ starting from $x(S_c)$ and $x(S_d)$) and $O(log(m))$ operations for line 26. All in all, this means that each iteration of the main cycle requires $O(m(n+\log(m)))$ operations in case A, which we can safely assume belongs to $O(mn)$ in all practical cases. This means that all occurrences of case A cumulatively contribute with $O(m^2n)$ operations to the final complexity.

As for case B, as previously mentioned if we do not enter in the body of the if clause of line 17 then for sure we entered in the body of the if clause of line 13; therefore, we only require $O(\log(m))$ operations for line 14 and a constant number of operations for all other lines. We can now argue that case B can only happen at most once for each time we executed line 26, which by itself already requires $O(\log(m))$ operations; therefore, we have already considered all possible contributes of case B.

Therefore, the complexity of the main cycle gets reduced from $O(m^3n)$ to $O(m^2n)$; since no other part of the algorithm requires more than this amount of operations, this results in an overall reduction of the complexity to $O(m^2n)$. While this may not sound exactly thrilling, in practice this means that the algorithm is now more than capable to handle real-world-sized data efficiently enough. Very similar optimizations can be carried on for the first two algorithms, resulting in the same final runtime.

Notice that we may still incur in a worst-case cost which is cubic in m if line 16 gets executed more than a constant amount of times, as we have previously assumed. However, if this is the case it means that most of the merges that we are performing are arbitrary, as in this situation there is no real evidence that the merged cells originally shared any mutation at all. This is clearly undesirable, and in practice it means that presumably we will not succeed in recovering the original clones. In other words, if the algorithm takes more than $O(m^2n)$ operations to complete we are not interested in the results anyway. We will experimentally see in Chapter 5 that this is actually the case in practice; furthermore, we will more in general verify that the so far derived time complexity bounds do hold.

```
Algorithm 4: OPTIMIZED_CLUSTER_ADJUSTED
Input: Mutation matrix $D$, list of cells $C$, list of mutations $V$.
Output: Dendrogram of clones and corresponding itemsets of mutations.
1  begin
2      $index \leftarrow 1$;   $\mathcal{C}_1 \leftarrow$ empty set;   $memo \leftarrow$ empty dictionary;
3      foreach cell $c_i \in C$ do
4          $S_i \leftarrow \{c_i\}$;   $\mathcal{C}_1 \leftarrow \mathcal{C}_1 \cup \{\, (S_i,\ x(S_i)) \,\}$;
5      $H \leftarrow$ empty max-heap;
6      foreach ordered pair $(S_a,\ x(S_a))$, $(S_b,\ x(S_b))$ from $\mathcal{C}_1$ do
7          if $x(S_a) \cap x(S_b) \neq \emptyset$ then
8              $\delta_{ab} \leftarrow \delta_w(x(S_a \cup S_b),\ S_a \cup S_b)$;
9              $\delta_{ab} \leftarrow \delta_{ab}$ − getExpVal($memo$, $|S_a \cup S_b|$, $C$, $V$, $D$);
10             $H$.append($\delta_{ab}$, $(S_a,\ x(S_a))$, $(S_b,\ x(S_b))$);
11     heapify($H$);
12     while not all cells are in a single cluster do
13         if $H$ is not empty then
14             $(S_a,\ x(S_a))$, $(S_b,\ x(S_b)) \leftarrow$ heappop($H$) ;
15         else
16             $(S_a,\ x(S_a))$, $(S_b,\ x(S_b)) \leftarrow$ elements of $\mathcal{C}_{index}$ such that
               $\delta_w(x(S_a \cup S_b),\ S_a \cup S_b)$ − getExpVal($memo$, $|S_a \cup S_b|$,
               $C$, $V$, $D$) is maximized;
17         if neither $(S_a,\ x(S_a))$ nor $(S_b,\ x(S_b))$ is marked as merged
           then
18             mark $(S_a,\ x(S_a))$ and $(S_b,\ x(S_b))$ as merged;
19             $S_c \leftarrow S_a \cup S_b$;
20             $\mathcal{C}_{index+1} \leftarrow \mathcal{C}_{index} \setminus \{\, (S_a,\ x(S_a)),\ (S_b,\ x(S_b)) \,\}$;
21             $\mathcal{C}_{index+1} \leftarrow \mathcal{C}_{index+1} \cup \{\, (S_c,\ x(S_c)) \,\}$;
22             foreach $(S_d,\ x(S_d)) \in \mathcal{C}_{index+1}$ do
23                 if $S_d \neq S_c \wedge x(S_d) \cap x(S_c) \neq \emptyset$ then
24                     $\delta_{cd} \leftarrow \delta_w(x(S_c \cup S_d),\ S_c \cup S_d)$;
25                     $\delta_{cd} \leftarrow \delta_{cd}$ − getExpVal($memo$, $|S_c \cup S_d|$, $C$,$V$,$D$);
26                     $H$.heappush($\delta_{cd}$, $(S_c,\ x(S_c))$, $(S_d,\ x(S_d))$);
27             $index \leftarrow index + 1$;
28     return $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m$;
```

4.4 Sampling efficiency

In Section 4.3 we assumed that the number of samples needed to estimate the expected value of the weighted density is a constant, and potentially a small one. This fact is mostly justified empirically, as we will see in Section 5.6; however, we can still propose some interesting, high-level theoretical considerations.

For simplicity, let us consider the unweighted density, for which we have derived more theoretical results. In particular, given any $k \in \mathbb{N}^+$ we can easily apply Propositions 4.1.2 and 4.1.4 to verify that an upper bound to the expected value of $\delta(x(S), S)$ is

$$\frac{1-p}{1-p^k}$$

when $S \sim \mathcal{U}(\{S \subseteq C \text{ s.t. } |S| = k\})$.

Similarly, it is trivial to verify that according to our choice of $x(S)$ the density of $\delta(x(S), S)$ can never be lower than

$$\frac{1}{|S|}$$

no matter the choice of S; hence, thanks to the linearity of the expected value we are assured that the expectation we are interested in is lower bounded by k^{-1}. We can actually verify an even stronger result: even the sampling approximation of the expected value is lower bounded by the same quantity, since as we have just seen each sample must individually be greater or equal to k^{-1}.

We can now plot both the upper and the lower bounds of our expectation as a function of the cardinality k of S, trying different values of p; this results in Figure 4.1. Notice that the upper bound of the expectation does not necessarily hold for its sampled approximation, too; and since we have no explicit formula for the variance, we do not even know how far from this upper bound we may reasonably get.

However, it is quite clear that as the value of p increases the upper bound of the correct expectation gets closer and closer to its lower bound; as seen in the Remark to Corollary 4.1.2.1, we even know that

$$\frac{1-p}{1-p^k} \to \frac{1}{k}$$

as $p \to 1$. Therefore, this is still a hint that we may expect the variability of our approximation to decrease as p increases. As a side note, notice how

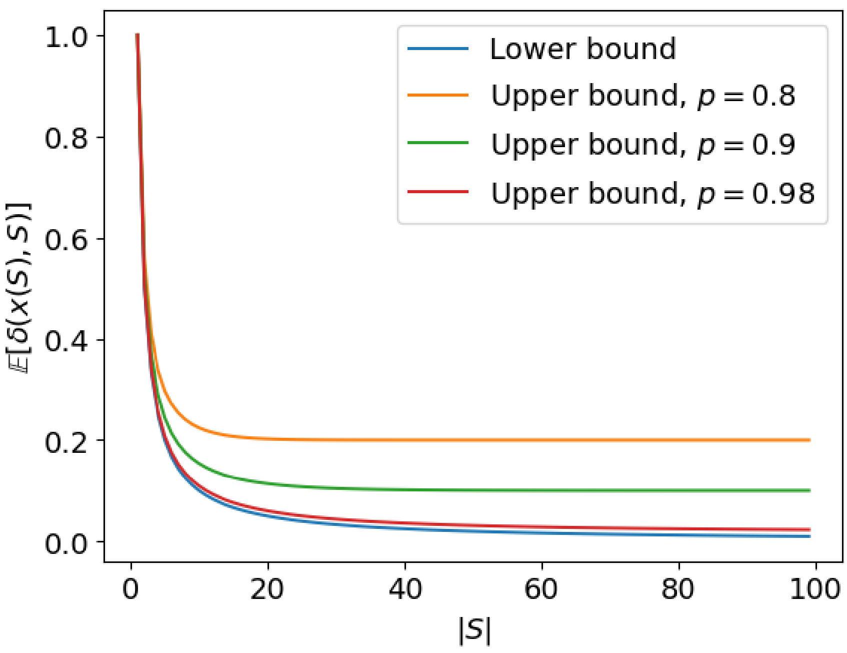

Figure 4.1: Bounds on the true expected density for various values of p.

the fact that the two bounds get closer and closer is very consistent with the fact that both intuitively and experimentally the problem gets harder as $p \to 1$.

Anyway, the most interesting consideration we can make is that the behaviour of the bounds depicted in Figure 4.1 does not depend upon the total number of cells, nor does it depend upon the total number of mutations. Once again, this is probably a hint to the fact that as we are assuming the number of samples needed to estimate the expected value is a constant with respect to the total size of our problem, especially for high values of p.

Chapter 5

Experimental Evaluation

In this Chapter we explore the capabilities of the proposed algorithms by discussing some experimental results. Such results are summarized as plots; the detailed numerical values can be found in Appendix A.

As discussed in Chapter 4, in order to compare the results with the ones from SMBClone we need a way to choose a single clustering among the ones in the dendrogram. The solution adopted is to simply choose the clustering with cardinality equal to the correct number of clones. While this approach could potentially give some advantage to the proposed algorithms, it allows us to evaluate their performance without any bias that would be introduced by choosing a specific model selection algorithm.

To better compare the results with the ones from [2] the same score is used throughout the experiments; namely, the Adjusted Rand Index (ARI). Given a set containing n elements and two different partitioning X and Y of such set, the Rand Index is defined as

$$\mathrm{RI} = \frac{\alpha}{\binom{n}{2}}$$

where α is the number of pairs of elements from the set that were either put in the same subset both in X and in Y or that were put in different subsets both in X and in Y. Therefore, the Rand Index measures the similarity between two different clusterings of the same input data. The Adjusted Rand Index additionally corrects for the expected value of such similarity, so that a value of one means that the correct clonal composition was perfectly recovered, while a value of zero corresponds to what we would get on average by randomly guessing; formally,

$$\mathrm{ARI} = \frac{\mathrm{RI} - \mathbb{E}[\mathrm{RI}]}{1 - \mathbb{E}[\mathrm{RI}]}.$$

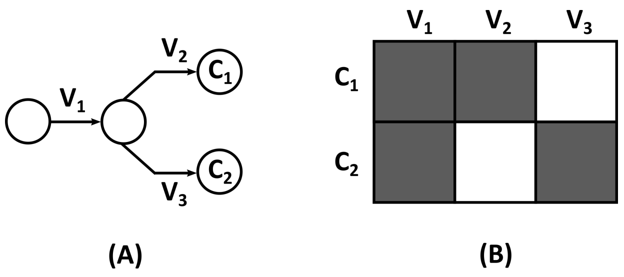

Figure 5.1: (**A**) The clone tree used in SBMClone and (**B**) its corresponding mutation matrix.

All synthetic samples were obtained using the code for simulated data generation from SBMClone; unless otherwise stated the parameters were chosen to be consistent with the ones used in [2], i. e. $m = 4000$, $n = 5000$, $p = 0.995$ and a mutation matrix as per Figure 5.1, with a fraction of overlapping mutations of 0.3. Since the noise strength is quite high in most tests, for ease of representation we will often refer to $\bar{p} \overset{\text{def}}{=} 1 - p$. To ensure that no bias was introduced through the data, the row and columns of the resulting matrices were randomly shuffled. Each result is averaged over five different matrices generated with the same parameters, unless otherwise stated. All tests were performed on a PC with a Intel® Centrino™ 2 dual-core processor and 2 GB of RAM. The code was implemented using Python, and a C++ porting is currently in its preliminary stages. Since currently the C++ version is not sufficiently refined, all presented results are relative to the Python implementation. Notice however that the code of SBMClone is mostly implemented in C++, so this may influence all runtime comparisons.

The Chapter is divided as follows: in Section 5.1 we will test different fractions of overlap between the mutations of the clones, highlighting the limitations of the basic density described in Section 4.1.2 and how the weighted density solves these problems. In Section 5.2 we will consider the effect of the number of cells on the algorithms; in particular, there will be a focus on the interesting behaviour of the proposed algorithms when few cells are present. Similarly, in Section 5.3 we will present the effect of varying the number of mutations, while for completeness we will test the effect of varying both the number of cells and the number of mutations at the same time in Section 5.4. Then, we will explore the consequences of more complex mutation trees on multiple problem sizes in Section 5.5. The considerations on the sampling efficiency presented during Section 4.4 will be experimentally tested in Section 5.6, confirming what we expected. A further discussion about the causes of the errors of the proposed algorithms will be the subject of Section 5.7; finally, we will test some data generated using realistic parameters in Section 5.8, where the proposed algorithms proved to perform better than SBMClone.

5.1 Varying the overlap

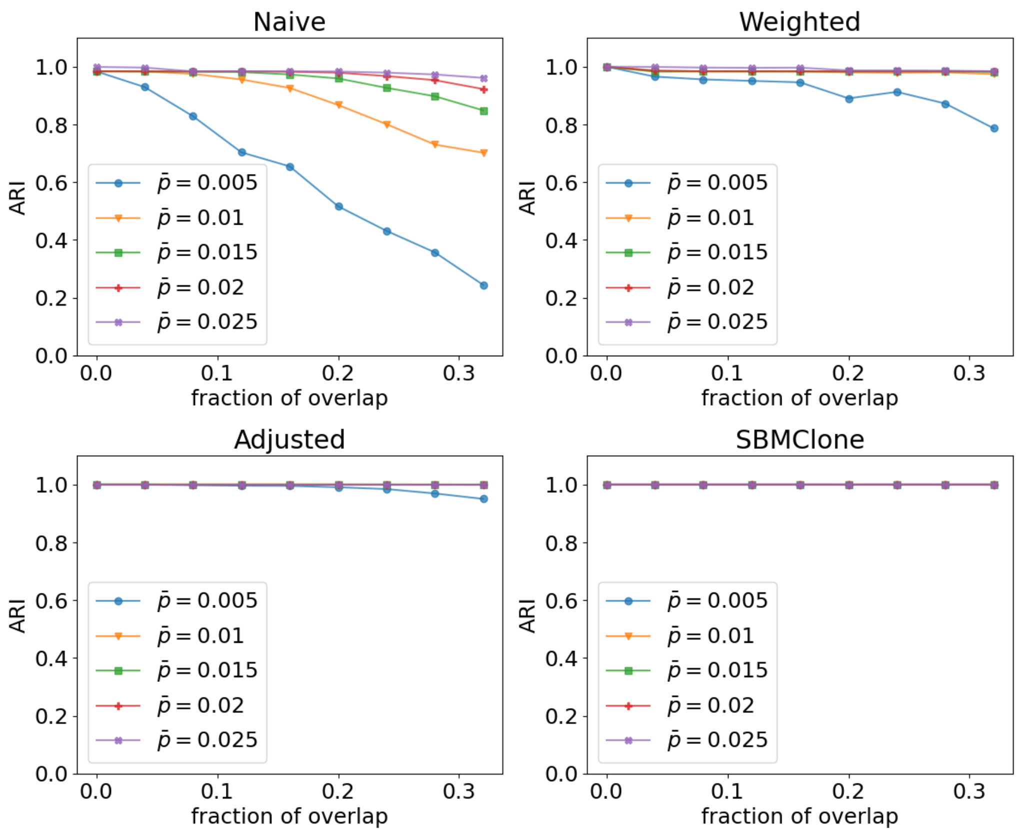

Figure 5.2: ARI vs. fraction of overlap.

The first test performed was checking the effect of the presence of a varying fraction of common mutations between the clones. In our case, the utilized mutation tree is the one of Figure 5.1; as such, the fraction of overlap is defined as $\frac{|V_1|}{n}$.

We have already discussed how the basic density may present some difficulties when several common mutations are present; Figure 5.2 does actually confirm this. In particular, we can verify how the naive algorithm struggles in retrieving the true clonal composition when the overlap increases, especially for low values of $\bar{p}$. On the bright side, the weighted version of the algorithm already shows a good performance for all noise strengths, and consistently achieves ARI equal to one when no overlap is present. This confirms that the introduction of the weighted density greatly mitigates the limitations of its vanilla version. Furthermore, while SBMClone itself clearly does not suffer from overlapping mutation groups at all, the adjusted version of the algorithm displays similar reconstruction capabilities in the vast majority of the considered parameter subspace.

As for runtimes, we can see in Figure 5.3 that all four algorithms required roughly comparable amounts of time, with SBMClone performing the best. Once again, this is at least partially due to the different programming language used in the implementations.

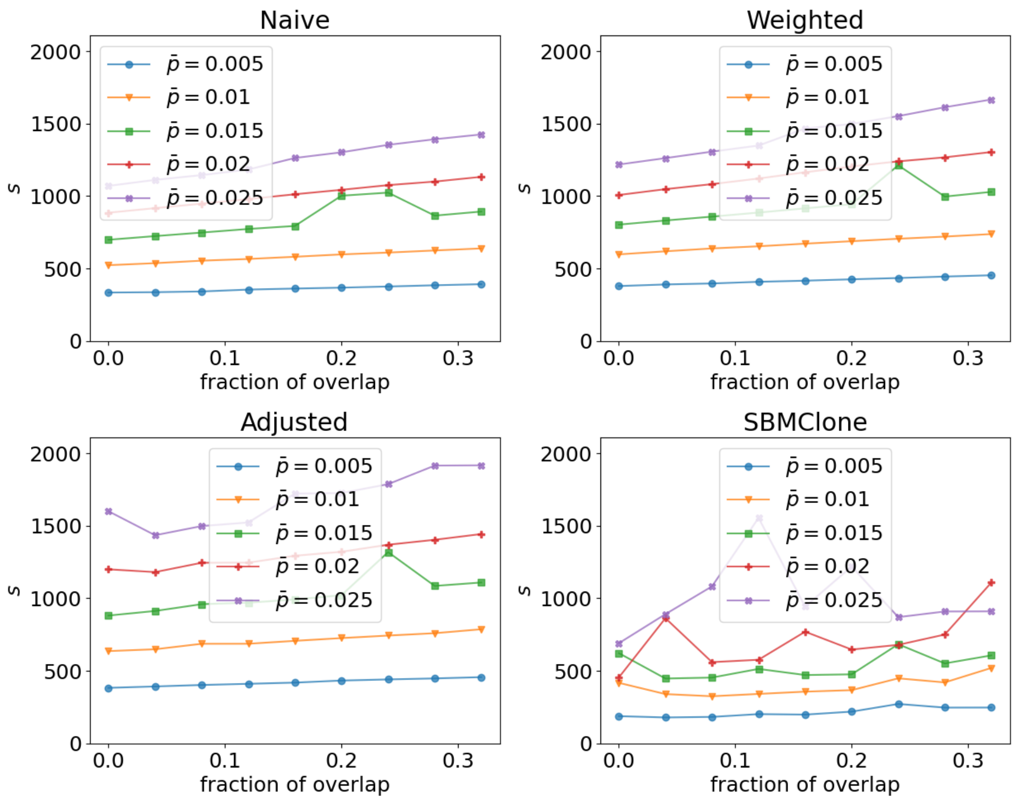

Figure 5.3: Runtime vs. fraction of overlap.

5.2 Varying the number of cells

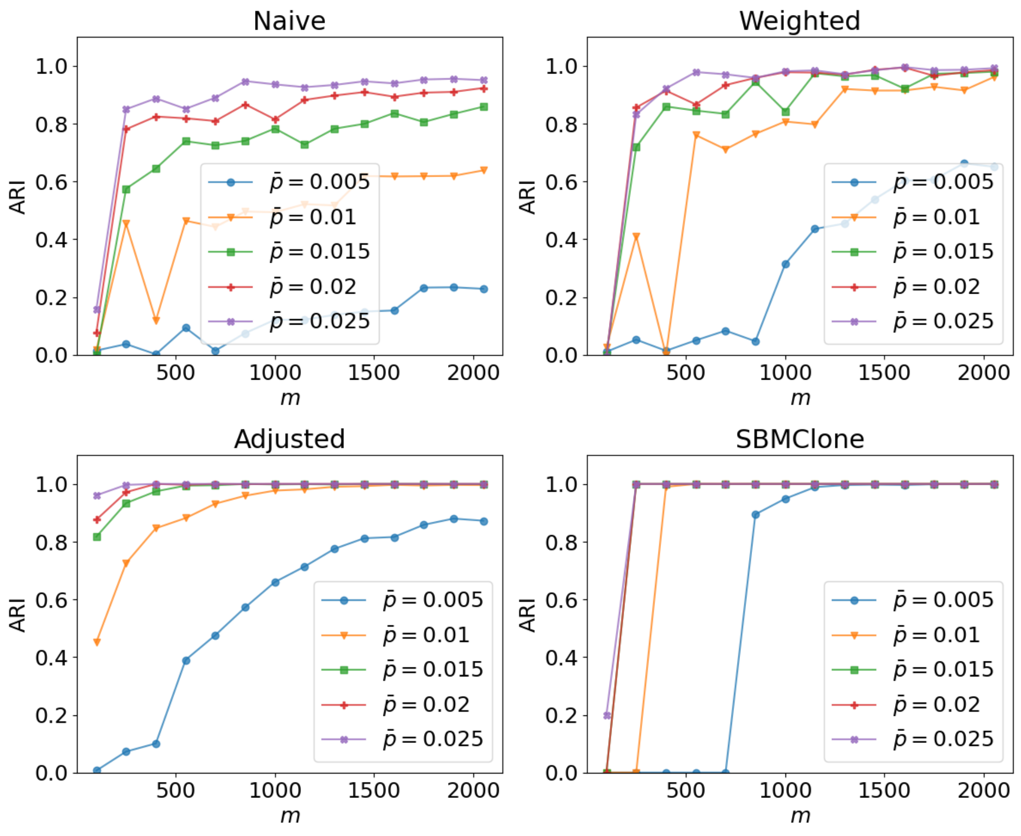

Figure 5.4: ARI vs. number of cells.

Another interesting test to perform is verifying how the algorithms behave as a function of the number of cells. The results are in Figure 5.4. Notice that both SBMClone and the adjusted algorithm start to perform perfectly in most cases for values of m as low as 1000, so reaching the usual value of $m = 4000$ was deemed unnecessary. Once again, we can see that the naive clustering algorithm performed the worst, while the weighted and adjusted versions reached increasingly better scores and SBMClone generally worked the best.

However, the adjusted clustering algorithm seems to often achieve results comparable to the ones of SBMClone, which is even consistently outperformed for lower values of $|C|$. This will be better discussed at the end of the Chapter.

Another interesting feature of the considered results is the presence of a local ARI maximum when a small number of cells is considered. Such local maximum can be identified both in the naive and the weighted version of the

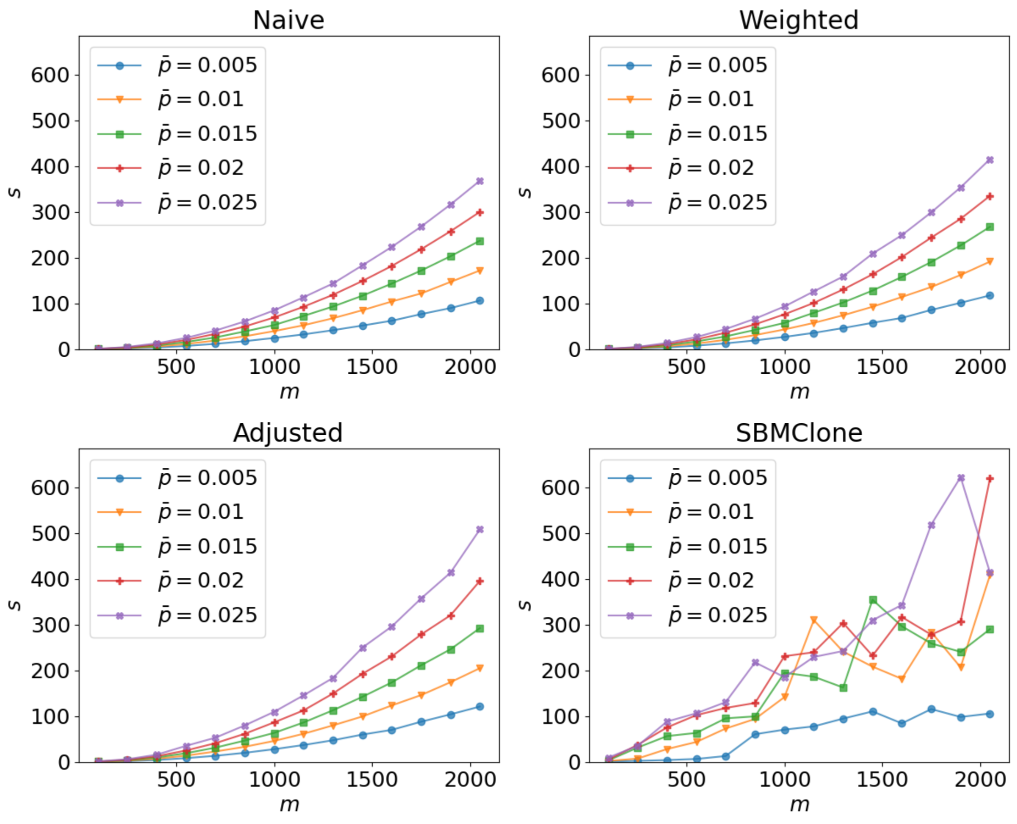

Figure 5.5: Runtime vs. number of cells.

clustering algorithm, while it is absent from the adjusted algorithm. The presence of this local maximum may be surprising, since we could expect the considered computational problem to get easier and easier as the number of cells (and thus the available data) increases. A possible explanation of this behaviour will be discussed in Section 5.2.1.

As for the runtimes, we can verify from Figure 5.5 that the theoretical predictions of Section 4.3 do hold in practice. In particular, all three clustering algorithms display a complexity quadratic in the number of cells, and the only difference between one another is a multiplicative constant factor. In any case, all runtimes are usually comparable, including the ones of SBMClone.

5.2.1 Dendrograms and distribution of the errors

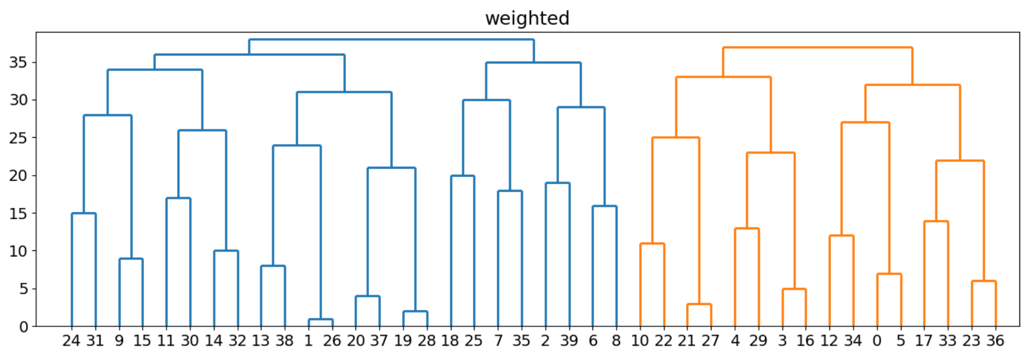

Figure 5.6: Dendrogram from the weighted algorithm, $m = 40$, $\bar{p} = 0.025$.

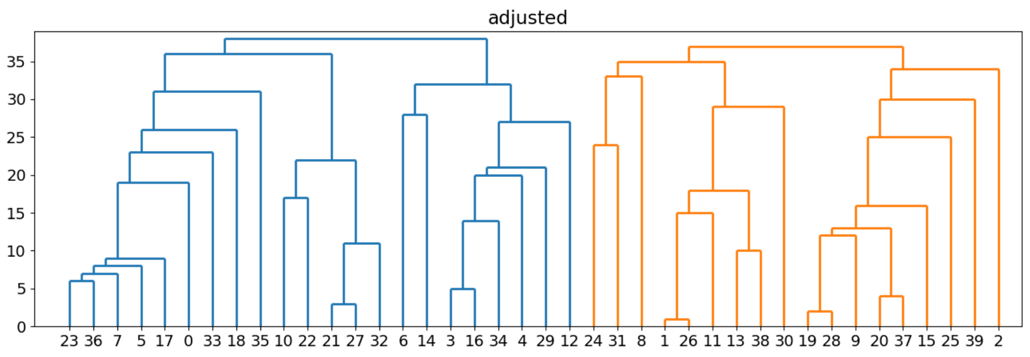

Figure 5.7: Dendrogram from the adjusted algorithm, $m = 40$, $\bar{p} = 0.025$.

To better understand how the number of cells influences the algorithms' performances we can directly look at some output dendrograms. The considered input mutation matrix was generated using noise strength $\bar{p} = 0.025$; to keep a good readability of the outputs, only $m = 40$ cells were considered.

The dendrogram from Figure 5.6 was generated using the weighted algorithm. Here we can clearly notice how the cells are merged following the clusters' sizes, i. e. at first all cells are grouped into pairs, then all pairs are merged into clusters of four cells and so on. This behaviour was expected, since as previously seen the expected density is a monotonically decreasing function of the number of cells in a cluster.

The dendrogram of the adjusted algorithm can be seen in Figure 5.7. This highlights a completely different behaviour, with cluster merged without any influence from their size. This means that the correction factor used in the adjusted density does actually help counteracting the bias introduced by the cardinality of the clusters.

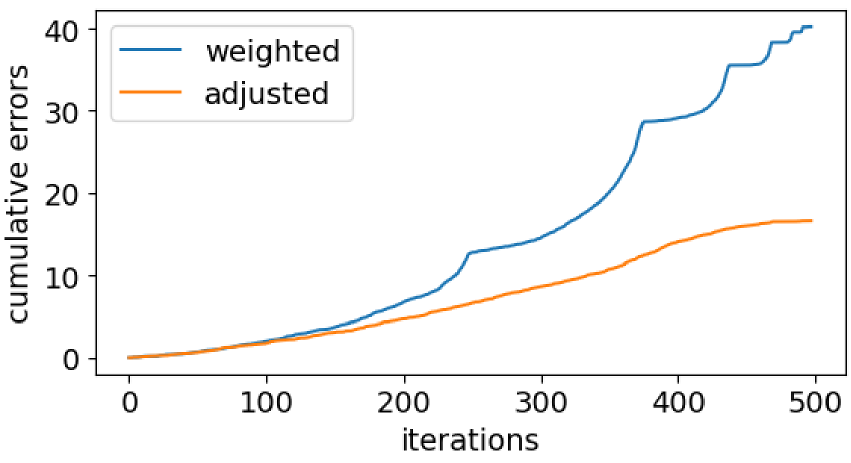

Figure 5.8: Cumulative errors vs. number of iterations.

The consequences of this structural difference between the two dendrograms can be seen in Figure 5.8. Here, the cumulative number of errors up to the $i-$th iteration is plotted for both the weighted and the adjusted algorithm. In this context, merging two clusters is considered to be an error whenever the majority of the cells from the first cluster originally belonged to a different clone with respect to the majority of the cells of the second cluster. The test was performed using $m = 500$, $\bar{p} = 0.01$ and averaging over 50 runs.

The consequence of merging clusters according to size in the weighted algorithm can be clearly seen: each time a given cardinality is considered the most promising clusters get merged at first, resulting in a somewhat slow increase in the number of cumulative errors. However, at some point only the most uncertain clusters are left; in this case, it is very likely that we may perform wrong merges, causing a steep increase in the number of cumulative errors. After all clusters of a given cardinality get processed the cycle repeats, until all cells get merged in a single cluster.

On the other hand, the adjusted algorithm tends to immediately exploit the most promising merges; then, as we proceed through the iterations, the problems becomes artificially easier, since the information from multiple cells is merged. This enables us to keep the most uncertain clusters for later, when we can merge even them with higher reliability due to the increased amount of information per cluster. This means that the increase of the cumulative number of errors is not only more gradual and uniform, but it also tends to slow down in the last iterations.

This different behaviour between the two algorithms clearly explains the difference in performance that was discussed in the previous Section. Furthermore, the local ARI maximum followed by a sudden decline of performance that was observed in both the naive and the weighted algorithms may be linked with the erratic behaviour of the errors distribution and the presence of abrupt increases in the amount of wrong merges.

5.3 Varying the number of mutations

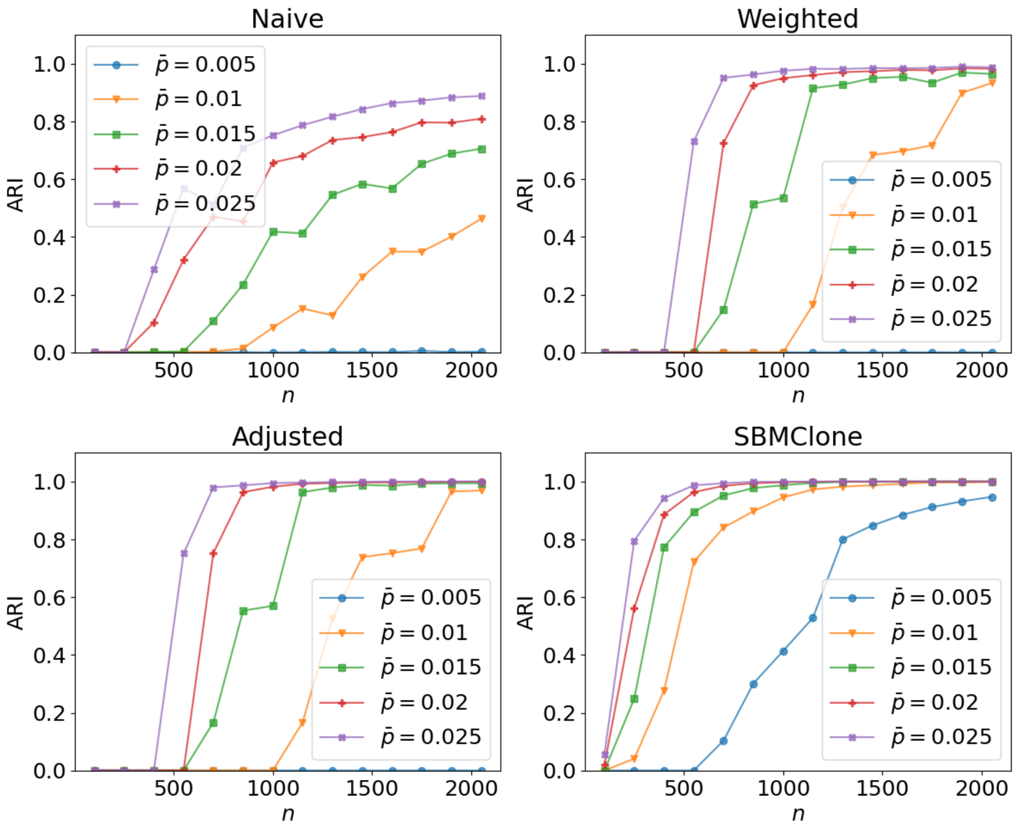

Figure 5.9: ARI vs. number of mutations.

We have previously analyzed how the algorithms respond to a change in the number of cells. Symmetrically, we will now consider the impact of a different number of mutations. In particular, the ARI reached by each algorithm is shown in Figure 5.9. As in the previous Section, reaching the usual value of $n = 5000$ was deemed unnecessary to highlight the interesting points. Generally speaking, the considerations presented with regards to the number of cells are still holding. The main difference is that in this case

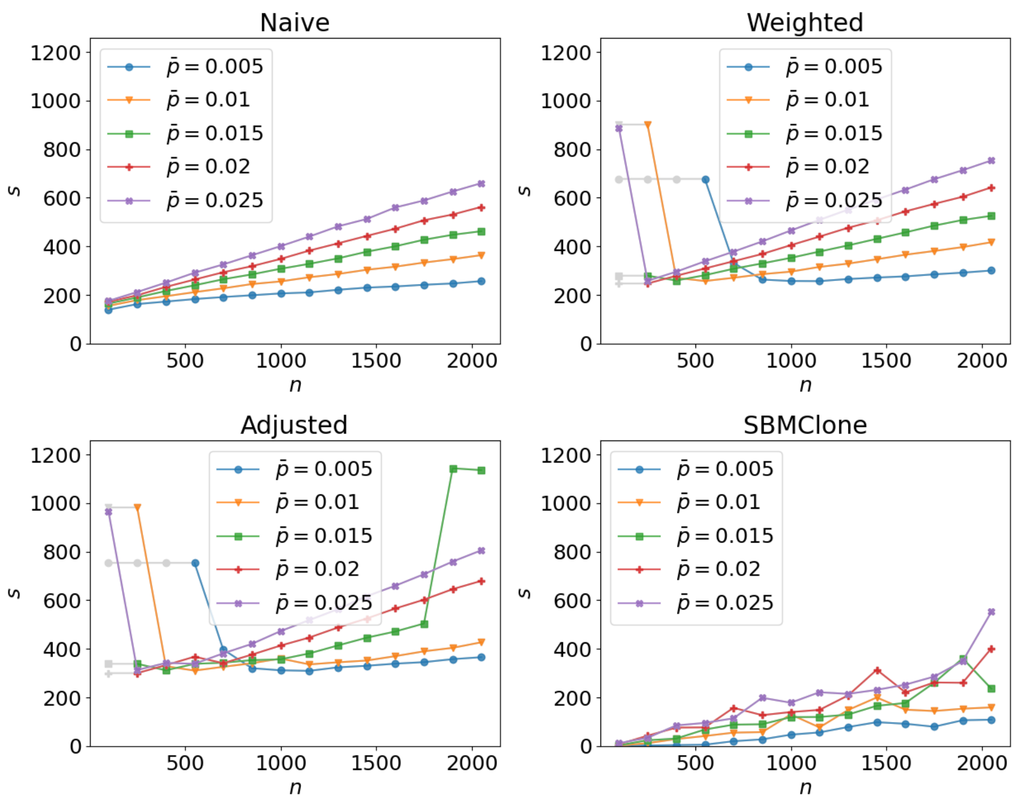

Figure 5.10: Runtime vs. number of mutations. Light gray points correspond to outliers which were clipped in the graphs.

a low number of mutations affects all algorithms much more than a low number of cells, especially when an higher amount of noise is present.

As for the runtimes (Figure 5.10), we can once again verify that the general trends correspond to the theoretical ones, with all runtimes linearly increasing with the number of mutations. However, while this holds true almost perfectly for the naive algorithm, the weighted and the adjusted versions present some outliers. As discussed in Section 4.3, this is caused by the worst-case scenario where the complexity is cubic in the number of cells. However, we can verify that this always results in an ARI very close to zero, meaning that the resulting output is not of interest.

5.4 Varying the number of both cells and mutations

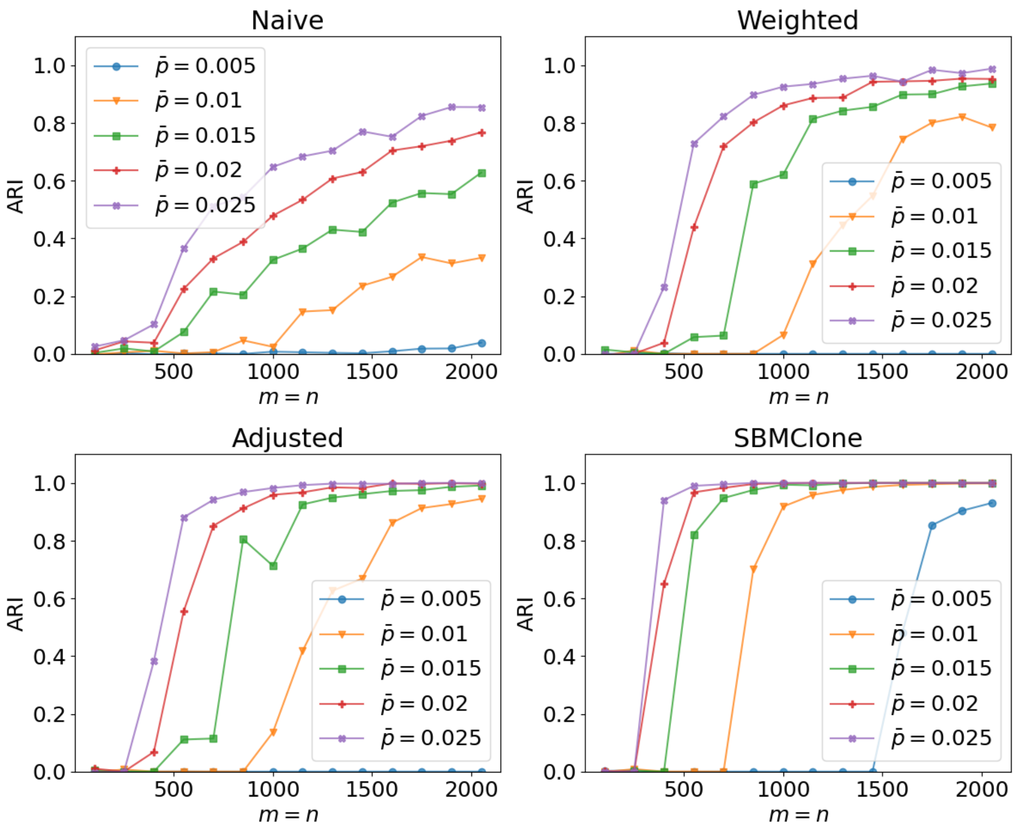

Figure 5.11: ARI vs. number of both cells and mutations.

For completeness, as a final test both the number of cells and the number of mutations was varied. This resulted in Figures 5.11 and 5.12. The results are consistent with what could be expected, in terms of both ARI and runtime, and reflect the same general considerations that were previously presented. The only interesting difference with respect to Section 5.3 is that this time there is not a dramatic increase in runtime when few mutations are present, mostly because now the corresponding low number of cells prevents the worst-case cubic scenario to heavily affect the overall time requirements.

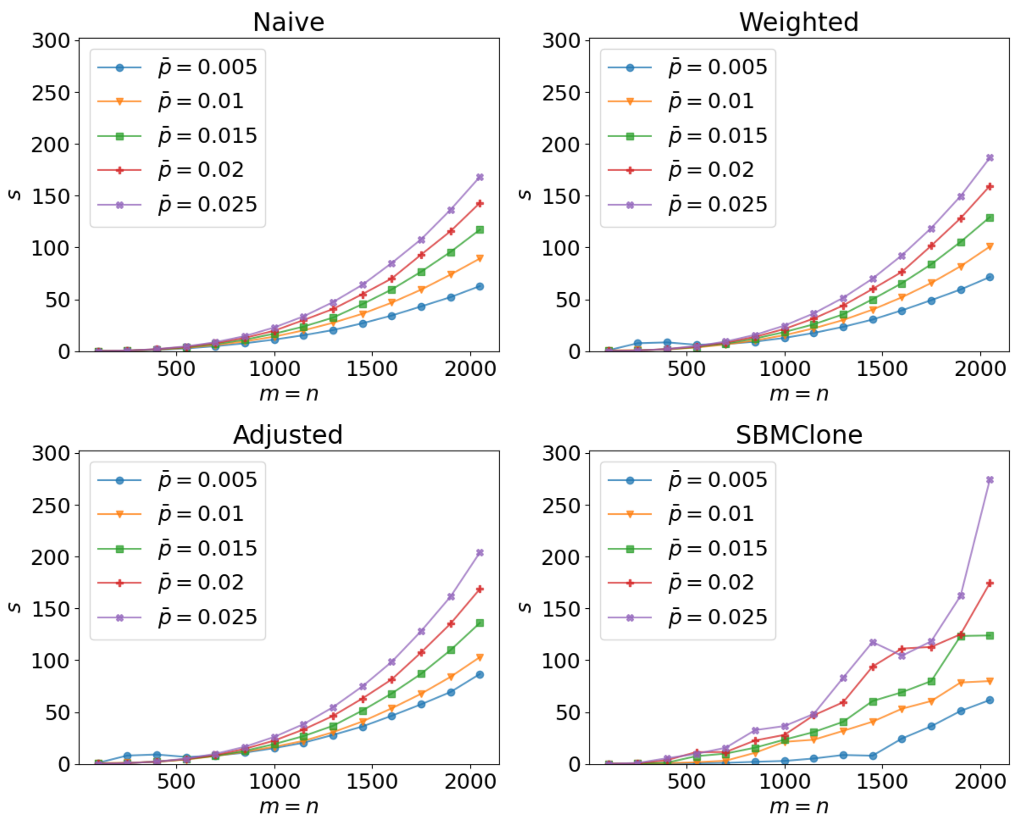

Figure 5.12: Runtime vs. number of both cells and mutations.

5.5 Varying the mutation tree

Up to now, all algorithms were tested on the same basic mutation tree. However, more complex structures may be found in practice; as such, the increasingly complex mutation trees represented in Figure 5.13 were considered. Multiple numbers of cells were tested: 4000 cells in total (Figure 5.14) 100 cells per clone (Figure 5.15) and 20 cells per clone (Figure 5.16). Notice that in this last case the number of mutations was increased to $50k$, since the results presented in the previous Sections suggested that all algorithms would probably fail with this few cells, unless more mutations are present. Furthermore, using $n = 50k$ is consistent with some real-world datasets, as discussed in Section 5.8.

In most cases, more complex underlying topologies led to lower ARI, as would be expected. There are some exceptions in the setups where the total number of cells was not constant: in these occasions, more elaborate mutation trees correspond to a larger number of cells; this may offset the additional complexity introduced by the topology.

The most interesting results are the ones with the lowest number of cells: here, consistently with what was seen in Section 5.2, the adjusted algorithm outperforms SBMClone for most topologies and most values of $\bar{p}$.

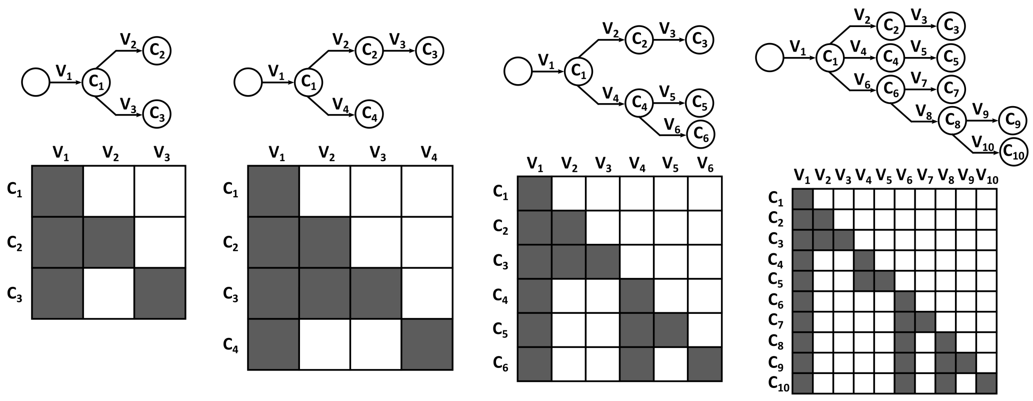

Figure 5.13: The four mutation trees tested, along with their corresponding matrices.

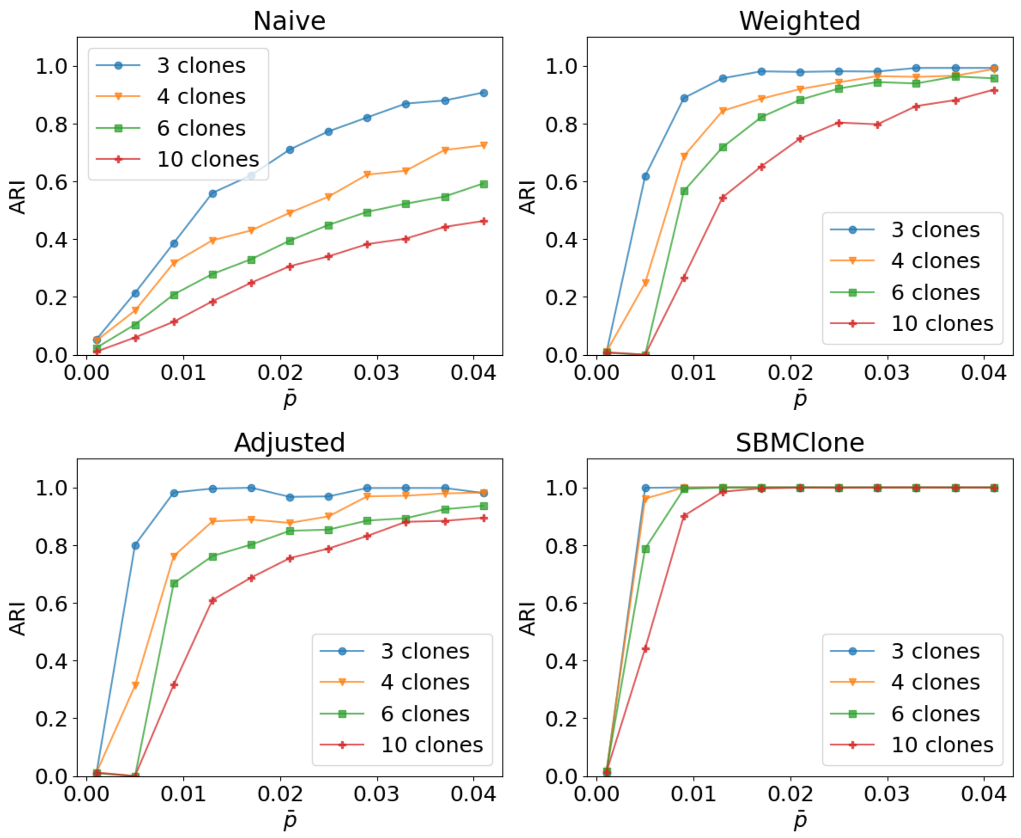

Figure 5.14: ARI vs. $\bar{p}$ for different mutation trees, $m = 4000$.

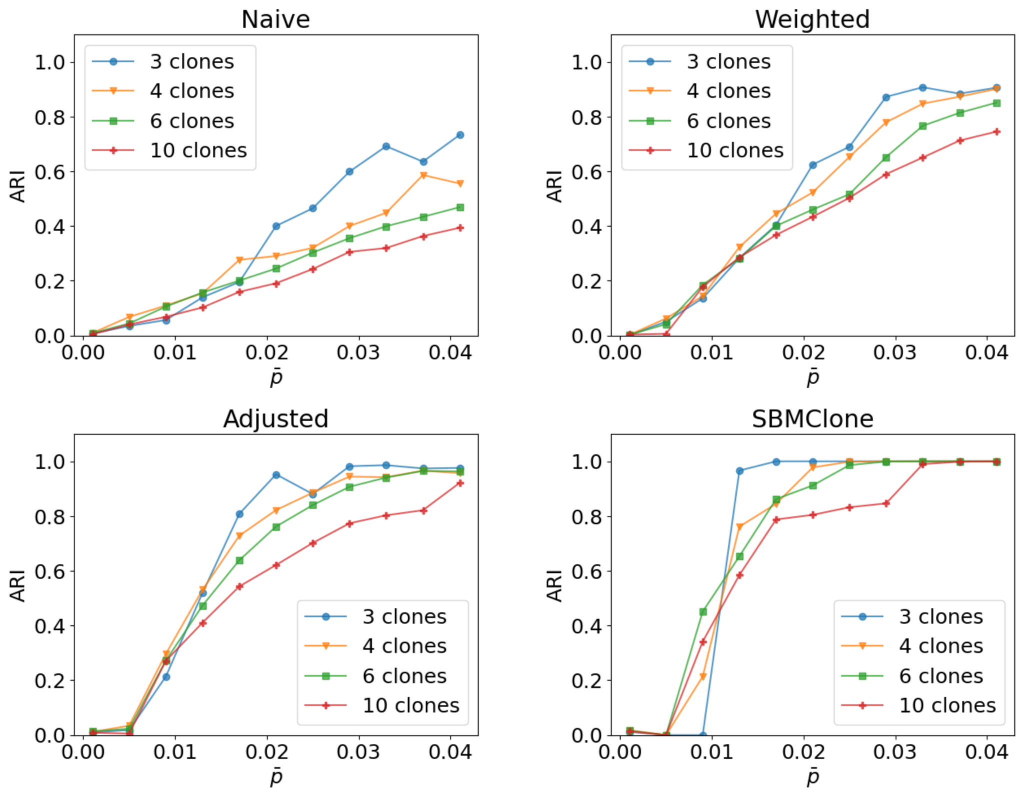

Figure 5.15: ARI vs. $\bar{p}$ for different mutation trees, 100 cells per clone.

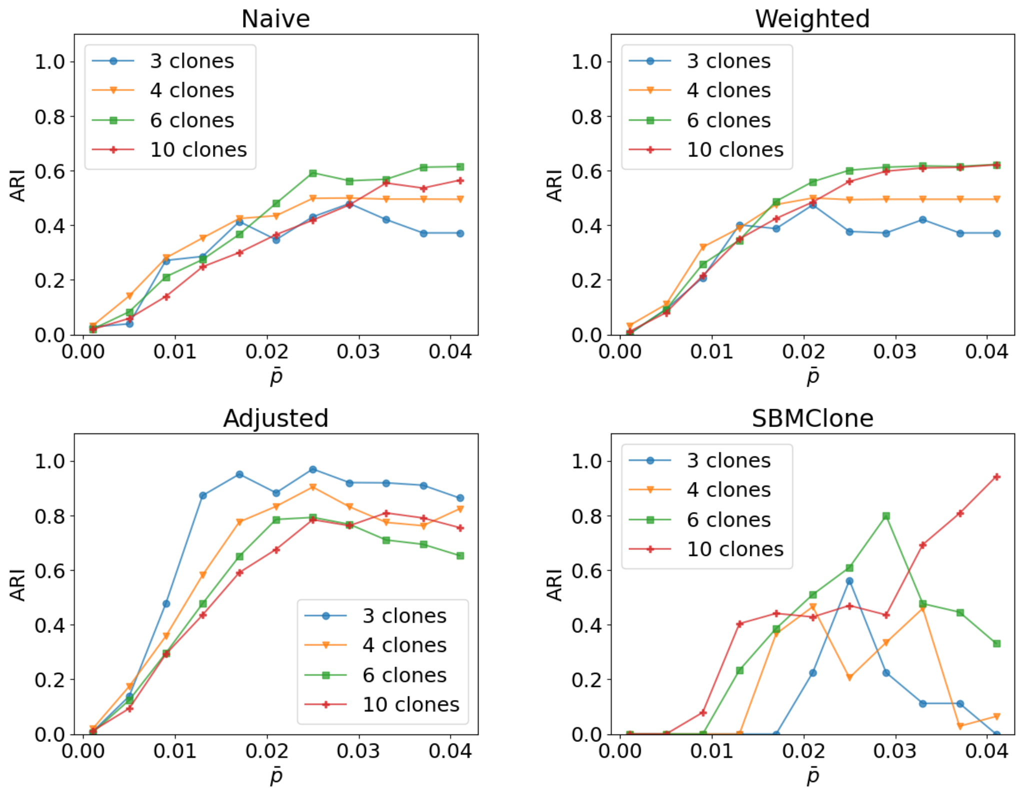

Figure 5.16: ARI vs. $\bar{p}$ for different mutation trees, 20 cells per clone.

5.6 Testing sampling efficiency

In Chapter 4 we have assumed that the adjusted algorithm could use a constant sample size, providing some hints about the reason why this could hold true. To verify this assumption empirically we can start from Figure 5.17, where the expected weighted density for $|S| = 2$ is computed for many different sample sizes. Looking at this plot it may seem like the variability of the approximation is quite considerable, despite it being slightly reduced for higher sample sizes.

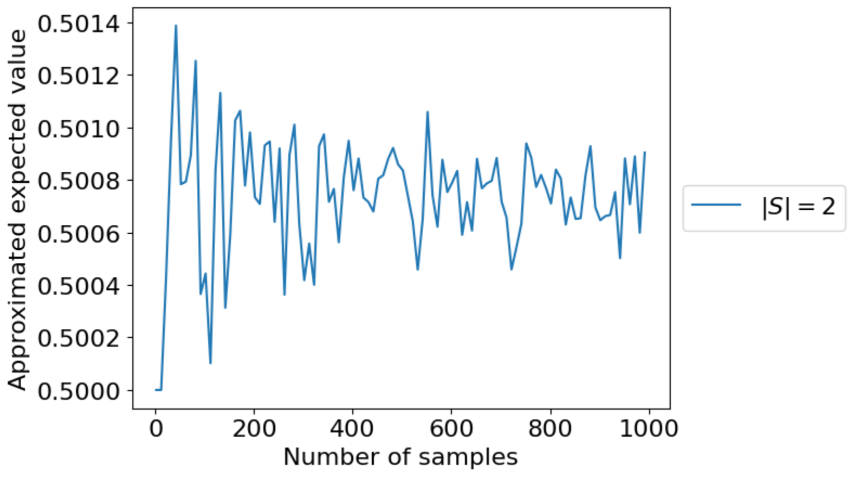

Figure 5.17: Detailed view of the estimated expected value for $|S| = 2$.

However, if we look at the global picture, we can realize that such variability is actually very small, even for the lowest sample size. In particular, Figure 5.18 clearly shows that the difference in magnitude between multiple values of $|S|$ overshadows the variability of the single approximation.

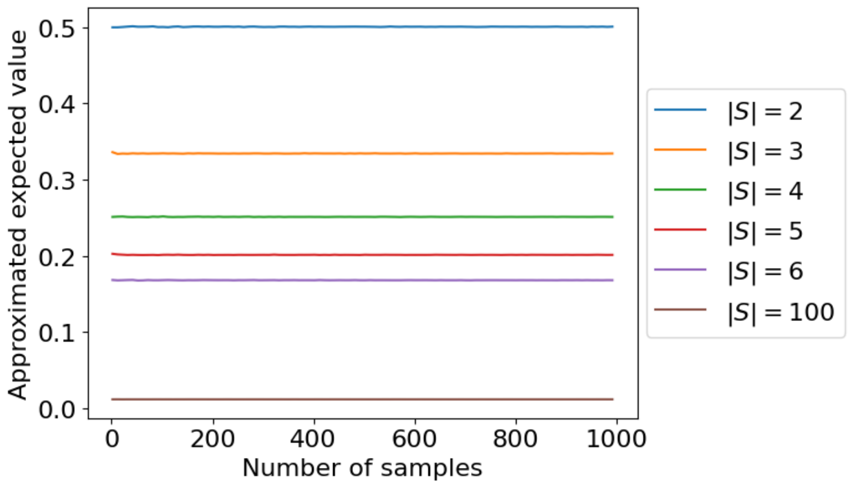

Figure 5.18: Global view of the estimations of different expected values.

A further confirmation of the ideas of Section 4.4 derives from Figure 5.19. Here $|S| = 20$ in all tests, and $\bar{p}$ is varied instead. We can then verify that as $\bar{p} \to 0$ the variability of the approximations decreases, which is exactly what we conjectured would happen due to the behaviour of the expected density bounds, as argued in the previous Chapter.

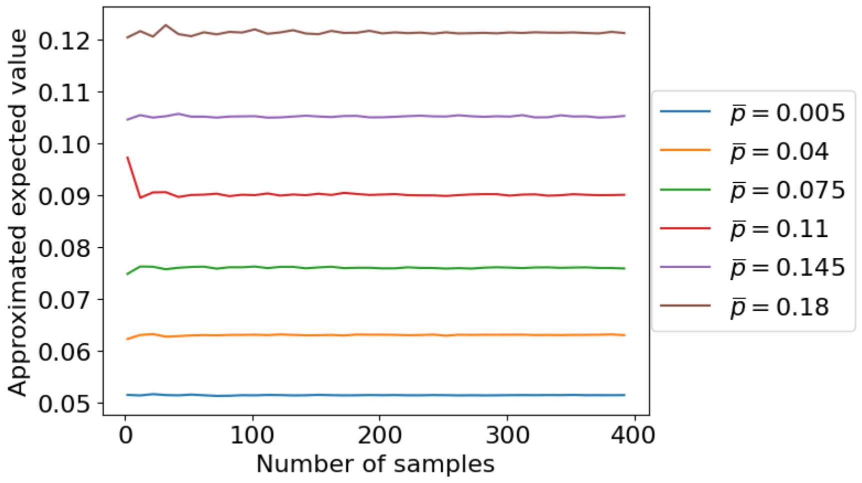

Figure 5.19: Estimated weighted density expected value, varying $\bar{p}$.

However, we have not answered the most important question yet: what happens to the adjusted algorithm when we vary the sample size? The answer lies in Figure 5.20. Here, we can verify that there is a noticeable benefit in terms of ARI when we go to a sample size of zero (meaning that we do not perform any sampling at all, making the adjusted algorithm an exact copy of the weighted algorithm) to a sample size of one. However, as the sample size increases, there is not any further benefit; on the other hand, the total runtime grows linearly with the sample size, as should be expected. All in all, this should definitively confirm our choice of fixing the sample size to be a small constant.

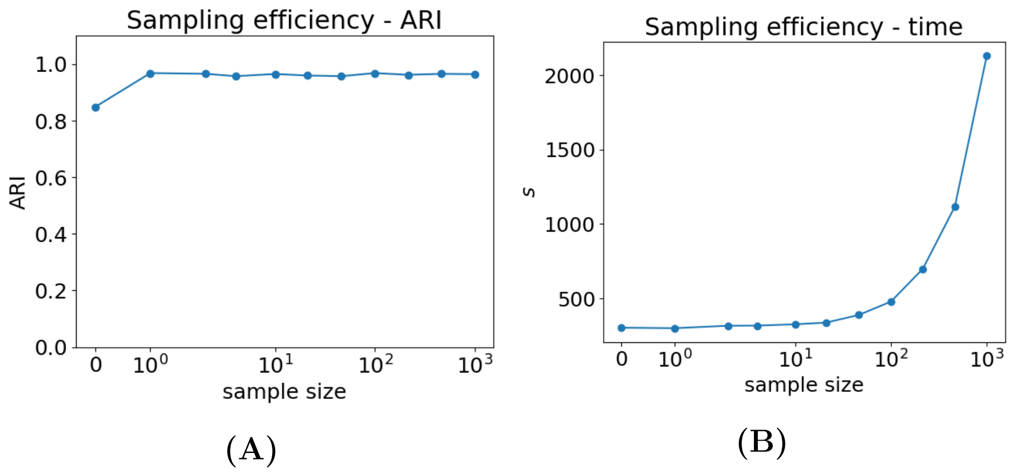

Figure 5.20: Average ARI **(A)** and runtime **(B)** for different sample sizes, symlog x axis scale.

5.7 Statistical and computational barriers

We have seen that the proposed algorithms can perform very well for some combinations of the problem's parameters; however, some other times the resulting ARI was not satisfactory. This naturally poses the question of whether we are encountering a statistical or a computational barrier. In the first case, data does not contain enough information to allow perfect recovery of the clonal composition using a density-based algorithm; in the second one, the information contained in the data could be sufficient but the proposed algorithms are not capable of retrieving the density-optimal partitioning.

A possible answer comes from Figure 5.21, where the adjusted algorithm was once again tested with a variable number of cells like we did in Section 5.2, using $\bar{p} = 0.005$. In this case, the weighted density of each of the two clones was averaged together and plotted along with the resulting ARI. In particular, the weighted density was computed for both the correct, original clones and for the ones returned by the algorithm.

As expected, both the correct and the estimated densities decrease for bigger values of m. However, the decrease in the estimated weighted density is more steep: the algorithm achieves an average density higher than the one of the correct clones for low values of m, but it starts to return a weighted density worse than the ground truth as the number of cells increases; how-

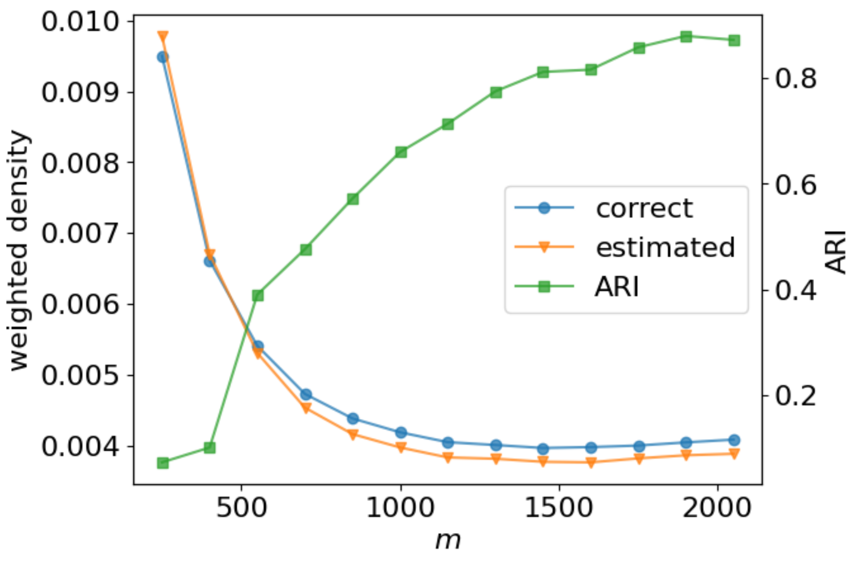

Figure 5.21: Relation between the density of the correct clones, the density of the estimated clones and the resulting ARI

ever, this results in higher ARI. All in all, this seems to suggest that we are initially dealing with a statistical barrier; then, as the available amount of data increases, density starts to correctly characterize the problem and the remaining amount of error is presumably imputable to the greedy choices of the algorithm.

Notice however that even when the barrier is mostly computational we continue to achieve better and better ARI as m increases, despite the difference between the correct and the estimated densities remaining almost constant. This suggests that density alone does not capture the whole picture: since it is only an heuristic score (despite being a provably meaningful one) finding the global optimum is not necessarily needed to better reconstruct clones; as such, we once again confirm that a greedy clustering algorithm can be suitable, being very efficient and enabling the exploitation of the more promising and informative merges, as previously discussed in Section 5.2.1.

5.8 Testing realistic parameters

As previously mentioned in Section 5.2 the clustering algorithms seem to outperform SBMClone when the cell count and $\bar{p}$ are low. Interestingly enough, the real-world datasets presented in [5] and tested by the authors of SBMClone [2] do contain a low cell count; in particular, each sample contains around $m = 45$ cells, $n = 50000$ mutations and a roughly estimated $\bar{p} \approx 0.02$. It was therefore all but natural to test all four algorithms on some synthetic data generated using these parameters; the mutation tree is the one of Figure 5.1. The results can be seen in Figures 5.22 and 5.23.

It is quite clear that in this setup all the proposed algorithms outperform SBMClone for most values of $\bar{p}$ with respect to both time and ARI. In particular, the adjusted algorithms consistently manages to achieve perfect or almost perfect identification of the correct clones starting from $\bar{p} \approx 0.01$, and therefore including the realistic case of $\bar{p} \approx 0.02$. On the other hand, SBMClone is quite far from returning a correct partitioning of the cells for most noise strengths, starting to provide slightly more meaningful results only when $\bar{p}$ is roughly twice as big as the estimated one.

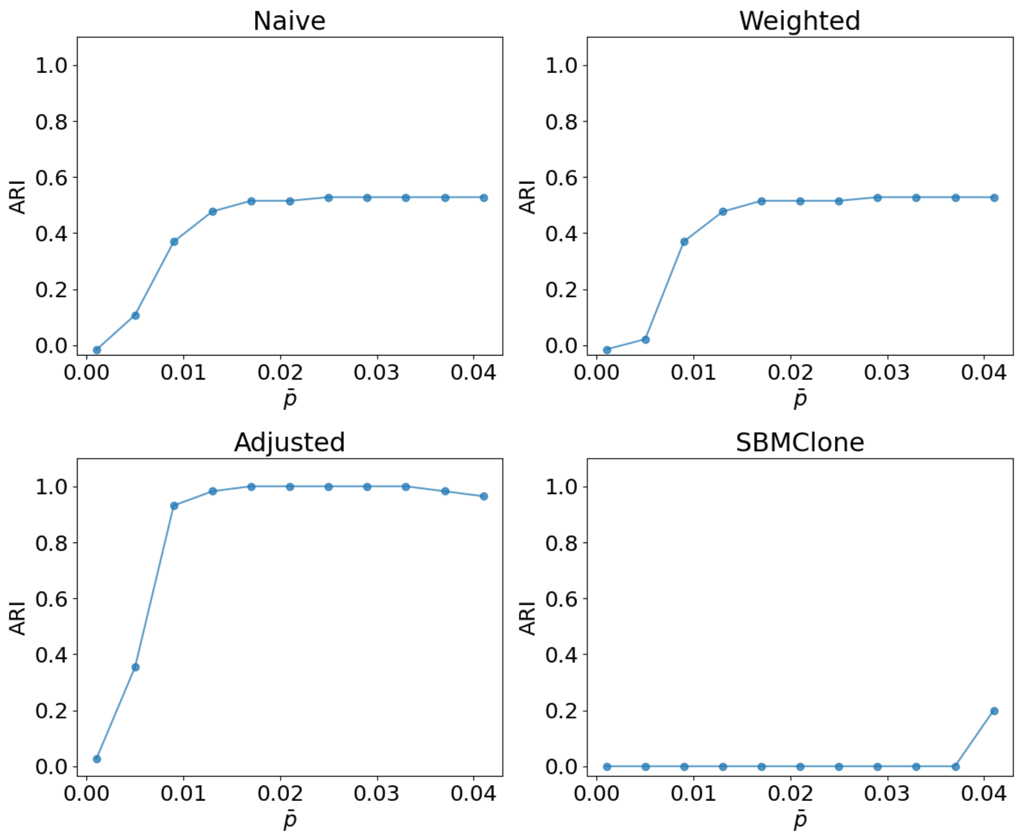

Figure 5.22: ARI vs. $\bar{p}$, realistic parameters.

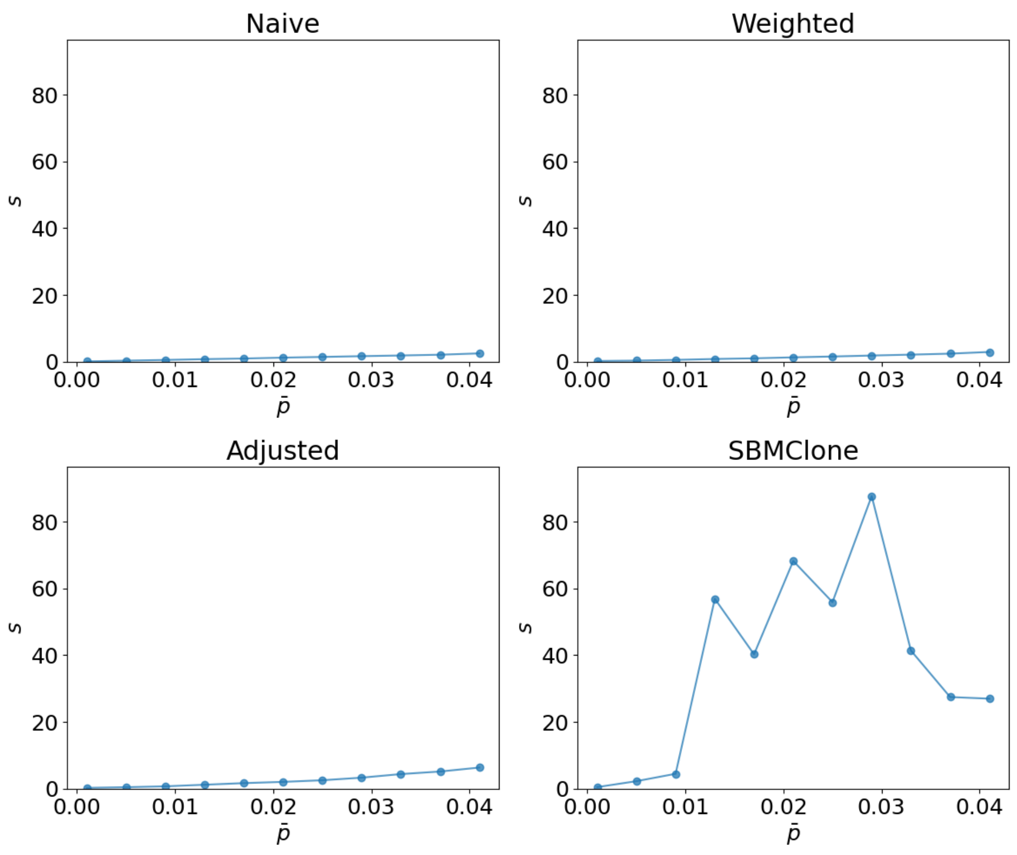

Figure 5.23: Runtime vs. $\bar{p}$, realistic parameters.

Chapter 6

Conclusions

To the best of our knowledge, in this work we presented the first algorithms based on noisy itemset mining that allows the recovery of the clonal composition of tumor samples from single nucleotide variations.

We theoretically proved the relationship between noisy itemsets, mutations and clones; this allowed the definition of a score to distinguish between true clones and random groups of cells. Extensive theoretical analyses proved that in expectation true clones are actually rewarded with a strictly higher score; furthermore, they enabled the definition of even more effective variants of the basic score.

In turns, this enabled us to develop and implement three greedy agglomerative clustering algorithms, whose runtimes were analyzed and consequently optimized. We could then perform comprehensive experiments on synthetic data, assessing the efficiency, the scalability and the accuracy of the proposed techniques and comparing them to a state of the art algorithm, SBMClone. Such tests also enabled us to better understand the inner workings of both the scores and the algorithms, from how the errors distribute along the iterations to what kind of statistical and computational barriers we may encounter in practice.

Incidentally, all three proposed algorithms displayed good recovery capabilities. Even the naive approach, despite its performance being inferior to SBMClone and the other proposed techniques, actually outperformed all the other algorithms tested in [2] whenever a direct comparison was possible.

We finally performed some tests on synthetic data generated using realistic parameters derived from [5] through [2]; in such conditions, the proposed approaches proved to outperform SBMClone. In particular, the adjusted algorithm seems to be particularly suitable to this kind of real-world data where lots of mutations are available and few cells are present: practical tests suggest that the reconstruction capability is mostly influenced by the number of available mutations; furthermore, since its complexity is linear in n, it remains quite efficient. Additionally, the greedy nature of the algorithm actually enables it to solve the easiest cases at first; this way, the most difficult decisions can be made at a later time, when the data is already partially merged and thus the signal to noise ratio is increased.

In light of these considerations, an interesting direction for future works would be to actually test the proposed algorithms on real datasets. Moreover, deriving some results on the variance of the proposed score could further improve our understanding of both the problem and the proposed solution; as such, this would be an interesting direction for future expansions of the work. Finally, from a more practical point of view it could be interesting to complete and refine the C++ implementation of the algorithms; preliminary tests on the current unoptimized version already display a five-fold increase in time efficiency, which would make the proposed approach quite competitive over SBMClone in a wider range of the input data parameter space.

Bibliography

[1] C. Gawad, W. Koh, and S. R. Quake, "Single-cell genome sequencing: current state of the science," *Nature Reviews Genetics*, vol. 17, pp. 175–188, Jan. 2016.

[2] M. A. Myers, S. Zaccaria, and B. J. Raphael, "Identifying tumor clones in sparse single-cell mutation data," *Bioinformatics*, vol. 36, pp. i186–i193, 07 2020.

[3] T. P. Peixoto, "Efficient monte carlo and greedy heuristic for the inference of stochastic block models," *Phys. Rev. E*, vol. 89, p. 012804, Jan 2014.

[4] T. P. Peixoto, "Hierarchical block structures and high-resolution model selection in large networks," *Phys. Rev. X*, vol. 4, p. 011047, Mar 2014.

[5] C. Kim, R. Gao, E. Sei, R. Brandt, J. Hartman, T. Hatschek, N. Crosetto, T. Foukakis, and N. E. Navin, "Chemoresistance evolution in triple-negative breast cancer delineated by single-cell sequencing," *Cell*, vol. 173, pp. 879–893.e13, May 2018.

[6] A. K. Casasent, M. Edgerton, and N. E. Navin, "Genome evolution in ductal carcinoma *in situ*: invasion of the clones," *The Journal of Pathology*, vol. 241, pp. 208–218, Nov. 2016.

[7] A. Davis and N. E. Navin, "Computing tumor trees from single cells," *Genome Biology*, vol. 17, May 2016.

[8] L. Melchor, A. Brioli, C. P. Wardell, A. Murison, N. E. Potter, M. F. Kaiser, R. A. Fryer, D. C. Johnson, D. B. Begum, S. H. Wilson, G. Vijayaraghavan, I. Titley, M. Cavo, F. E. Davies, B. A. Walker, and G. J. Morgan, "Single-cell genetic analysis reveals the composition of initiating clones and phylogenetic patterns of branching and parallel evolution in myeloma," *Leukemia*, vol. 28, pp. 1705–1715, Jan. 2014.

[9] Z. Yu, A. Li, and M. Wang, "CloneCNA: detecting subclonal somatic copy number alterations in heterogeneous tumor samples from whole-exome sequencing data," *BMC Bioinformatics*, vol. 17, Aug. 2016.

[10] R. Wang, D.-Y. Lin, and Y. Jiang, “SCOPE: A normalization and copy-number estimation method for single-cell DNA sequencing,” *Cell Systems*, vol. 10, pp. 445–452.e6, May 2020.

[11] S. Hui and R. Nielsen, “SCONCE: a method for profiling copy number alterations in cancer evolution using single-cell whole genome sequencing,” *Bioinformatics*, vol. 38, pp. 1801–1808, Jan. 2022.

[12] S. Salehi, A. Steif, A. Roth, S. Aparicio, A. Bouchard-Côté, and S. P. Shah, “ddClone: joint statistical inference of clonal populations from single cell and bulk tumour sequencing data,” *Genome Biology*, vol. 18, Mar. 2017.

[13] G. Satas, S. Zaccaria, G. Mon, and B. J. Raphael, “SCARLET: Single-cell tumor phylogeny inference with copy-number constrained mutation losses,” *Cell Systems*, vol. 10, pp. 323–332.e8, Apr. 2020.

[14] S. Malikic, F. R. Mehrabadi, S. Ciccolella, M. K. Rahman, C. Ricketts, E. Haghshenas, D. Seidman, F. Hach, I. Hajirasouliha, and S. C. Sahinalp, “PhISCS: a combinatorial approach for subperfect tumor phylogeny reconstruction via integrative use of single-cell and bulk sequencing data,” *Genome Research*, vol. 29, pp. 1860–1877, Oct. 2019.

[15] K. Yuan, T. Sakoparnig, F. Markowetz, and N. Beerenwinkel, “BitPhylogeny: a probabilistic framework for reconstructing intra-tumor phylogenies,” *Genome Biology*, vol. 16, Feb. 2015.

[16] K. Jahn, J. Kuipers, and N. Beerenwinkel, “Tree inference for single-cell data,” *Genome Biology*, vol. 17, May 2016.

[17] J. Singer, J. Kuipers, K. Jahn, and N. Beerenwinkel, “Single-cell mutation identification via phylogenetic inference,” *Nature Communications*, vol. 9, Dec. 2018.

[18] H. Zafar, N. Navin, K. Chen, and L. Nakhleh, “SiCloneFit: Bayesian inference of population structure, genotype, and phylogeny of tumor clones from single-cell genome sequencing data,” *Genome Research*, vol. 29, pp. 1847–1859, Oct. 2019.

[19] N. Borgsmüller, J. Bonet, F. Marass, A. Gonzalez-Perez, N. Lopez-Bigas, and N. Beerenwinkel, “BnpC: Bayesian non-parametric cluster-

ing of single-cell mutation profiles," *Bioinformatics*, vol. 36, pp. 4854–4859, June 2020.

[20] E. M. Ross and F. Markowetz, "OncoNEM: inferring tumor evolution from single-cell sequencing data," *Genome Biology*, vol. 17, Apr. 2016.

[21] Z. Yu, F. Du, and L. Song, "SCClone: Accurate clustering of tumor single-cell DNA sequencing data," *Frontiers in Genetics*, vol. 13, Jan. 2022.

[22] A. Roth, A. McPherson, E. Laks, J. Biele, D. Yap, A. Wan, M. A. Smith, C. B. Nielsen, J. N. McAlpine, S. Aparicio, A. Bouchard-Côté, and S. P. Shah, "Clonal genotype and population structure inference from single-cell tumor sequencing," *Nature Methods*, vol. 13, pp. 573–576, May 2016.

[23] S. Ciccolella, C. Ricketts, M. S. Gomez, M. Patterson, D. Silverbush, P. Bonizzoni, I. Hajirasouliha, and G. D. Vedova, "Inferring cancer progression from single-cell sequencing while allowing mutation losses," *Bioinformatics*, vol. 37, pp. 326–333, Aug. 2020.

[24] M. El-Kebir, "SPhyR: tumor phylogeny estimation from single-cell sequencing data under loss and error," *Bioinformatics*, vol. 34, pp. i671–i679, Sept. 2018.

[25] Z. Yu, H. Liu, F. Du, and X. Tang, "GRMT: Generative reconstruction of mutation tree from scratch using single-cell sequencing data," *Frontiers in Genetics*, vol. 12, June 2021.

[26] Z. Chen, F. Gong, L. Wan, and L. Ma, "RobustClone: a robust PCA method for tumor clone and evolution inference from single-cell sequencing data," *Bioinformatics*, vol. 36, pp. 3299–3306, Mar. 2020.

[27] Z. Yu and F. Du, "AMC: accurate mutation clustering from single-cell DNA sequencing data," *Bioinformatics*, vol. 38, pp. 1732–1734, Dec. 2021.

[28] J. Liu, S. Paulsen, X. Sun, W. Wang, A. B. Nobel, and J. Prins, "Mining approximate frequent itemsets in the presence of noise: Algorithm and analysis," in *SDM*, 2006.

[29] X. Sun and A. B. Nobel, "On the size and recovery of submatrices of ones in a random binary matrix," *Journal of Machine Learning Research*, vol. 9, no. 80, pp. 2431–2453, 2008.

[30] C. Yang, U. Fayyad, and P. S. Bradley, "Efficient discovery of error-tolerant frequent itemsets in high dimensions," in *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, (New York, NY, USA), p. 194–203, Association for Computing Machinery, 2001.

[31] J. Liu, S. Paulsen, W. Wang, A. Nobel, and J. Prins, "Mining approximate frequent itemsets from noisy data," in *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pp. 4 pp.–, 2005.

[32] H. Cheng, P. S. Yu, and J. Han, "Ac-close: Efficiently mining approximate closed itemsets by core pattern recovery," in *Sixth International Conference on Data Mining (ICDM'06)*, pp. 839–844, 2006.

[33] H. Cheng, P. S. Yu, and J. Han, "Approximate frequent itemset mining in the presence of random noise," in *Soft Computing for Knowledge Discovery and Data Mining* (O. Maimon and L. Rokach, eds.), pp. 363–389, Springer, 2008.

[34] K. Mouhoubi, L. Letocart, and C. Rouveirol, "Itemset mining in noisy contexts: A hybrid approach," in *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, pp. 33–40, 2011.

Appendix A

Detailed experimental results

This appendix contains all the numerical results presented in Chapter 5, together with their standard deviations. Each time an algorithm was the best performing one in terms of either ARI or runtime, the corresponding entry is highlighted in boldface.

A.1 Overlap

Algorithm: Naive					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	0.0	0.984	0.001	334.219	14.756
0.005	0.04	0.929	0.017	336.535	2.259
0.005	0.08	0.828	0.018	341.511	2.226
0.005	0.12	0.703	0.034	354.454	3.731
0.005	0.16	0.654	0.018	361.489	3.408
0.005	0.2	0.516	0.044	367.888	3.337
0.005	0.24	0.431	0.069	375.663	2.665
0.005	0.28	0.357	0.068	384.227	2.663
0.005	0.32	0.244	0.041	391.747	2.845

Table A.1: Detailed data for Naive - $\bar{p} = 0.005$.

Algorithm: Naive					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	0.0	0.984	0.000	522.865	2.245
0.01	0.04	0.983	0.002	536.824	1.566
0.01	0.08	0.975	0.004	553.971	2.579
0.01	0.12	0.956	0.004	565.723	5.090
0.01	0.16	0.926	0.011	580.971	2.477
0.01	0.2	0.867	0.032	597.268	3.166
0.01	0.24	0.801	0.029	609.967	2.806
0.01	0.28	0.730	0.023	624.401	2.704
0.01	0.32	0.702	0.022	638.597	0.473

Table A.2: Detailed data for Naive - $\bar{p} = 0.010$.

Algorithm: Naive					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	0.0	0.984	0.000	698.293	2.572
0.015	0.04	0.983	0.001	723.616	4.285
0.015	0.08	0.982	0.001	748.061	4.901
0.015	0.12	0.981	0.002	772.827	1.612
0.015	0.16	0.973	0.006	793.652	2.100
0.015	0.2	0.959	0.006	1002.476	410.058
0.015	0.24	0.927	0.021	1023.687	402.779
0.015	0.28	0.898	0.014	864.926	5.291
0.015	0.32	0.849	0.023	893.063	10.181

Table A.3: Detailed data for Naive - $\bar{p} = 0.015$.

Algorithm: Naive					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	0.0	0.984	0.000	885.247	3.113
0.02	0.04	0.984	0.000	916.485	1.219
0.02	0.08	0.984	0.001	947.736	1.842
0.02	0.12	0.984	0.000	978.824	2.046
0.02	0.16	0.982	0.002	1012.464	8.917
0.02	0.2	0.979	0.001	1043.023	2.534
0.02	0.24	0.967	0.004	1075.429	5.778
0.02	0.28	0.953	0.012	1099.897	2.711
0.02	0.32	0.922	0.016	1132.664	6.725

Table A.4: Detailed data for Naive - $\bar{p} = 0.020$.

Algorithm: Naive					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	0.0	0.999	0.001	1070.209	3.694
0.025	0.04	0.997	0.007	1111.708	7.307
0.025	0.08	0.984	0.001	1144.678	5.613
0.025	0.12	0.984	0.000	**1183.155**	3.819
0.025	0.16	0.984	0.000	1262.837	25.349
0.025	0.2	0.983	0.001	1301.968	15.680
0.025	0.24	0.979	0.002	1353.389	7.270
0.025	0.28	0.973	0.003	1392.115	12.235
0.025	0.32	0.961	0.002	1425.284	38.148

Table A.5: Detailed data for Naive - $\bar{p} = 0.025$.

Algorithm: Weighted					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	0.0	**1.000**	0.000	378.851	13.518
0.005	0.04	0.966	0.011	389.837	1.721
0.005	0.08	0.956	0.033	396.807	4.686
0.005	0.12	0.951	0.004	408.054	0.801
0.005	0.16	0.946	0.019	416.025	1.990
0.005	0.2	0.890	0.062	425.295	2.637
0.005	0.24	0.913	0.046	434.375	2.645
0.005	0.28	0.872	0.062	444.369	2.232
0.005	0.32	0.786	0.031	453.319	1.806

Table A.6: Detailed data for Weighted - $\bar{p} = 0.005$.

Algorithm: Weighted					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	0.0	**1.000**	0.000	597.512	2.061
0.01	0.04	0.984	0.001	618.648	2.982
0.01	0.08	0.983	0.001	638.549	1.728
0.01	0.12	0.983	0.001	653.330	3.010
0.01	0.16	0.983	0.001	671.473	3.287
0.01	0.2	0.980	0.004	688.397	4.074
0.01	0.24	0.979	0.002	705.259	1.616
0.01	0.28	0.980	0.003	720.217	2.971
0.01	0.32	0.974	0.007	737.646	2.310

Table A.7: Detailed data for Weighted - $\bar{p} = 0.010$.

Algorithm: Weighted					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	0.0	**1.000**	0.000	803.028	1.232
0.015	0.04	0.983	0.001	831.379	3.811
0.015	0.08	0.984	0.000	857.961	3.763
0.015	0.12	0.984	0.000	886.092	1.412
0.015	0.16	0.984	0.000	915.180	4.389
0.015	0.2	0.983	0.001	943.498	3.753
0.015	0.24	0.984	0.000	1212.213	542.013
0.015	0.28	0.983	0.001	996.461	6.468
0.015	0.32	0.982	0.002	1029.012	4.486

Table A.8: Detailed data for Weighted - $\bar{p} = 0.015$.

Algorithm: Weighted					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	0.0	**1.000**	0.000	1007.411	1.908
0.02	0.04	0.987	0.007	1047.654	7.410
0.02	0.08	0.984	0.000	1082.395	1.242
0.02	0.12	0.984	0.000	1121.082	4.927
0.02	0.16	0.984	0.000	1164.380	4.519
0.02	0.2	0.984	0.000	1204.203	7.812
0.02	0.24	0.984	0.001	1239.521	12.978
0.02	0.28	0.984	0.000	1267.259	5.915
0.02	0.32	0.984	0.000	1303.802	7.206

Table A.9: Detailed data for Weighted - $\bar{p} = 0.020$.

Algorithm: Weighted					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	0.0	**1.000**	0.000	1217.754	7.399
0.025	0.04	0.999	0.002	1261.907	2.154
0.025	0.08	0.997	0.007	1306.634	5.538
0.025	0.12	0.996	0.007	1348.641	5.399
0.025	0.16	0.997	0.007	1466.508	40.727
0.025	0.2	0.987	0.007	1497.494	6.296
0.025	0.24	0.987	0.007	1551.839	5.534
0.025	0.28	0.986	0.006	1613.016	6.157
0.025	0.32	0.984	0.000	1666.336	11.171

Table A.10: Detailed data for Weighted - $\bar{p} = 0.025$.

Algorithm: Adjusted					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	0.0	**1.000**	0.000	382.835	12.023
0.005	0.04	1.000	0.000	392.304	2.358
0.005	0.08	0.998	0.002	402.428	5.663
0.005	0.12	0.996	0.005	410.489	4.937
0.005	0.16	0.995	0.002	418.784	1.475
0.005	0.2	0.991	0.003	433.023	1.287
0.005	0.24	0.984	0.003	440.811	3.886
0.005	0.28	0.969	0.008	447.609	2.014
0.005	0.32	0.950	0.010	456.654	3.227

Table A.11: Detailed data for Adjusted - $\bar{p} = 0.005$.

Algorithm: Adjusted					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	0.0	**1.000**	0.000	636.111	15.295
0.01	0.04	**1.000**	0.000	648.695	5.577
0.01	0.08	**1.000**	0.000	686.507	19.269
0.01	0.12	**1.000**	0.000	686.631	3.467
0.01	0.16	**1.000**	0.000	706.700	4.134
0.01	0.2	1.000	0.000	725.737	3.353
0.01	0.24	0.999	0.001	743.132	4.754
0.01	0.28	0.999	0.001	759.441	7.220
0.01	0.32	0.999	0.001	786.686	2.406

Table A.12: Detailed data for Adjusted - $\bar{p} = 0.010$.

Algorithm: Adjusted					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	0.0	**1.000**	0.000	881.148	39.254
0.015	0.04	**1.000**	0.000	912.627	18.658
0.015	0.08	**1.000**	0.000	959.591	31.213
0.015	0.12	**1.000**	0.000	969.715	9.528
0.015	0.16	**1.000**	0.000	991.685	18.425
0.015	0.2	**1.000**	0.000	1021.567	10.215
0.015	0.24	**1.000**	0.000	1318.351	578.225
0.015	0.28	**1.000**	0.000	1085.329	20.506
0.015	0.32	**1.000**	0.000	1108.931	6.919

Table A.13: Detailed data for Adjusted - $\bar{p} = 0.015$.

Algorithm: Adjusted					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	0.0	**1.000**	0.000	1199.882	150.436
0.02	0.04	**1.000**	0.000	1180.654	29.839
0.02	0.08	**1.000**	0.000	1245.062	43.716
0.02	0.12	**1.000**	0.000	1245.656	24.003
0.02	0.16	**1.000**	0.000	1293.239	36.717
0.02	0.2	**1.000**	0.000	1320.704	23.426
0.02	0.24	**1.000**	0.000	1369.350	20.508
0.02	0.28	**1.000**	0.000	1403.423	42.109
0.02	0.32	**1.000**	0.000	1443.009	26.693

Table A.14: Detailed data for Adjusted - $\bar{p} = 0.020$.

Algorithm: Adjusted					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	0.0	**1.000**	0.000	1600.157	428.212
0.025	0.04	**1.000**	0.000	1434.740	50.537
0.025	0.08	**1.000**	0.000	1497.932	40.864
0.025	0.12	**1.000**	0.000	1522.601	47.454
0.025	0.16	**1.000**	0.000	1721.825	104.582
0.025	0.2	**1.000**	0.000	1725.390	19.563
0.025	0.24	**1.000**	0.000	1786.862	35.002
0.025	0.28	**1.000**	0.000	1914.997	87.578
0.025	0.32	**1.000**	0.000	1916.654	21.942

Table A.15: Detailed data for Adjusted - $\bar{p} = 0.025$.

Algorithm: SBMClone					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	0.0	**1.000**	0.000	**187.633**	8.310
0.005	0.04	**1.000**	0.000	**178.554**	5.819
0.005	0.08	**1.000**	0.000	**182.460**	8.213
0.005	0.12	**1.000**	0.000	**201.607**	7.773
0.005	0.16	**1.000**	0.000	**198.299**	6.735
0.005	0.2	**1.000**	0.000	**218.681**	35.990
0.005	0.24	**1.000**	0.000	**271.662**	86.091
0.005	0.28	**1.000**	0.000	**246.556**	28.476
0.005	0.32	**1.000**	0.000	**246.945**	4.660

Table A.16: Detailed data for SBMClone - $\bar{p} = 0.005$.

Algorithm: SBMClone					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	0.0	**1.000**	0.000	**417.276**	244.905
0.01	0.04	**1.000**	0.000	**339.706**	73.663
0.01	0.08	**1.000**	0.000	**324.782**	2.968
0.01	0.12	**1.000**	0.000	**340.790**	11.048
0.01	0.16	**1.000**	0.000	**356.519**	10.331
0.01	0.2	**1.000**	0.000	**366.832**	16.497
0.01	0.24	**1.000**	0.000	**448.270**	96.836
0.01	0.28	**1.000**	0.000	**420.160**	11.621
0.01	0.32	**1.000**	0.000	**519.981**	38.536

Table A.17: Detailed data for SBMClone - $\bar{p} = 0.010$.

Algorithm: SBMClone					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	0.0	**1.000**	0.000	**621.870**	105.123
0.015	0.04	**1.000**	0.000	**446.912**	25.151
0.015	0.08	**1.000**	0.000	**453.560**	28.528
0.015	0.12	**1.000**	0.000	**513.244**	80.172
0.015	0.16	**1.000**	0.000	**471.043**	16.060
0.015	0.2	**1.000**	0.000	**475.802**	18.989
0.015	0.24	**1.000**	0.000	**683.423**	348.424
0.015	0.28	**1.000**	0.000	**551.078**	52.199
0.015	0.32	**1.000**	0.000	**606.408**	29.595

Table A.18: Detailed data for SBMClone - $\bar{p} = 0.015$.

Algorithm: SBMClone					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	0.0	**1.000**	0.000	**454.712**	14.896
0.02	0.04	**1.000**	0.000	**862.936**	246.471
0.02	0.08	**1.000**	0.000	**559.874**	22.581
0.02	0.12	**1.000**	0.000	**576.338**	12.850
0.02	0.16	**1.000**	0.000	**769.846**	341.196
0.02	0.2	**1.000**	0.000	**646.679**	38.051
0.02	0.24	**1.000**	0.000	**679.063**	12.294
0.02	0.28	**1.000**	0.000	**750.228**	48.124
0.02	0.32	**1.000**	0.000	**1108.609**	195.871

Table A.19: Detailed data for SBMClone - $\bar{p} = 0.020$.

Algorithm: SBMClone					
$\bar{p}$	overlap	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	0.0	**1.000**	0.000	**690.449**	64.931
0.025	0.04	**1.000**	0.000	**890.716**	35.189
0.025	0.08	**1.000**	0.000	**1082.332**	236.912
0.025	0.12	**1.000**	0.000	1554.307	833.641
0.025	0.16	**1.000**	0.000	**942.718**	198.334
0.025	0.2	**1.000**	0.000	**1219.163**	556.504
0.025	0.24	**1.000**	0.000	**870.048**	23.995
0.025	0.28	**1.000**	0.000	**909.167**	105.779
0.025	0.32	**1.000**	0.000	**909.998**	22.262

Table A.20: Detailed data for SBMClone - $\bar{p} = 0.025$.

A.2 Number of cells

Algorithm: Naive					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	100	**0.015**	0.044	**0.203**	0.006
0.005	250	0.037	0.049	**1.306**	0.108
0.005	400	0.002	0.005	**3.555**	0.141
0.005	550	0.093	0.041	6.977	0.155
0.005	700	0.014	0.016	**11.581**	0.187
0.005	850	0.075	0.052	**17.387**	0.288
0.005	1000	0.121	0.064	**24.432**	0.297
0.005	1150	0.122	0.068	**32.197**	0.258
0.005	1300	0.138	0.059	**41.346**	0.521
0.005	1450	0.149	0.039	**51.481**	0.652
0.005	1600	0.153	0.065	**62.013**	2.923
0.005	1750	0.232	0.064	**76.470**	0.687
0.005	1900	0.233	0.042	**89.681**	0.737
0.005	2050	0.228	0.037	106.135	1.723

Table A.21: Detailed data for Naive - $\bar{p} = 0.005$.

Algorithm: Naive					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	100	0.016	0.044	**0.329**	0.004
0.01	250	0.453	0.093	**2.215**	0.077
0.01	400	0.118	0.152	**5.810**	0.111
0.01	550	0.464	0.084	**11.293**	0.074
0.01	700	0.442	0.076	**18.753**	0.358
0.01	850	0.496	0.036	**28.042**	0.175
0.01	1000	0.493	0.059	**39.367**	0.460
0.01	1150	0.521	0.073	**52.139**	0.262
0.01	1300	0.517	0.057	**67.471**	0.577
0.01	1450	0.618	0.067	**84.966**	0.185
0.01	1600	0.617	0.074	**104.068**	1.115
0.01	1750	0.618	0.043	**121.600**	3.265
0.01	1900	0.619	0.041	**146.804**	1.123
0.01	2050	0.638	0.043	**171.833**	0.884

Table A.22: Detailed data for Naive - $\bar{p} = 0.010$.

Algorithm: Naive					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	100	-0.002	0.007	0.494	0.077
0.015	250	0.575	0.058	**3.005**	0.090
0.015	400	0.644	0.085	**7.972**	0.069
0.015	550	0.739	0.039	**15.574**	0.157
0.015	700	0.725	0.038	**25.623**	0.269
0.015	850	0.740	0.049	**38.588**	0.266
0.015	1000	0.783	0.037	**52.639**	2.170
0.015	1150	0.727	0.026	**72.535**	0.362
0.015	1300	0.782	0.039	**93.064**	0.356
0.015	1450	0.799	0.034	**116.640**	0.655
0.015	1600	0.836	0.019	**143.132**	1.254
0.015	1750	0.805	0.021	**171.637**	0.894
0.015	1900	0.833	0.020	**202.966**	1.955
0.015	2050	0.858	0.028	**236.905**	1.360

Table A.23: Detailed data for Naive - $\bar{p} = 0.015$.

Algorithm: Naive					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	100	0.076	0.184	0.631	0.069
0.02	250	0.780	0.072	**3.787**	0.056
0.02	400	0.824	0.035	**10.225**	0.158
0.02	550	0.818	0.015	**20.052**	0.080
0.02	700	0.809	0.025	**33.402**	0.208
0.02	850	0.867	0.038	**49.477**	0.291
0.02	1000	0.815	0.049	**69.211**	0.394
0.02	1150	0.882	0.013	**92.658**	0.411
0.02	1300	0.897	0.008	**118.699**	1.191
0.02	1450	0.909	0.021	**148.769**	0.479
0.02	1600	0.893	0.013	**181.669**	1.047
0.02	1750	0.907	0.037	**218.188**	1.086
0.02	1900	0.910	0.022	**257.447**	6.134
0.02	2050	0.923	0.018	**299.858**	7.733

Table A.24: Detailed data for Naive - $\bar{p} = 0.020$.

Algorithm: Naive					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	100	0.156	0.226	**0.734**	0.010
0.025	250	0.850	0.062	**4.744**	0.050
0.025	400	0.887	0.016	**12.585**	0.081
0.025	550	0.851	0.062	**24.586**	0.261
0.025	700	0.890	0.060	**40.490**	0.296
0.025	850	0.947	0.008	**60.571**	0.447
0.025	1000	0.936	0.034	**84.781**	0.632
0.025	1150	0.926	0.041	**113.319**	0.383
0.025	1300	0.934	0.034	**143.397**	3.876
0.025	1450	0.946	0.018	**182.918**	3.106
0.025	1600	0.939	0.029	**223.206**	0.995
0.025	1750	0.953	0.003	**267.617**	1.112
0.025	1900	0.955	0.010	**315.880**	2.113
0.025	2050	0.950	0.018	**367.945**	2.143

Table A.25: Detailed data for Naive - $\bar{p} = 0.025$.

Algorithm: Weighted					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	100	0.010	0.027	0.253	0.096
0.005	250	0.052	0.056	1.461	0.113
0.005	400	0.014	0.021	3.824	0.152
0.005	550	0.050	0.039	7.619	0.150
0.005	700	0.083	0.126	12.644	0.249
0.005	850	0.047	0.033	19.013	0.204
0.005	1000	0.314	0.108	26.684	0.102
0.005	1150	0.435	0.046	35.506	0.437
0.005	1300	0.454	0.051	46.057	0.380
0.005	1450	0.538	0.154	57.512	0.640
0.005	1600	0.602	0.049	68.152	3.353
0.005	1750	0.606	0.111	85.609	0.764
0.005	1900	0.662	0.108	101.198	0.961
0.005	2050	0.651	0.074	117.397	1.442

Table A.26: Detailed data for Weighted - $\bar{p} = 0.005$.

Algorithm: Weighted					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	100	0.025	0.026	0.340	0.003
0.01	250	0.410	0.154	2.254	0.117
0.01	400	-0.000	0.002	6.172	0.112
0.01	550	0.760	0.011	12.174	0.094
0.01	700	0.711	0.122	20.365	0.254
0.01	850	0.764	0.058	30.596	0.267
0.01	1000	0.807	0.019	43.018	0.290
0.01	1150	0.797	0.064	57.289	0.455
0.01	1300	0.920	0.014	73.917	0.209
0.01	1450	0.914	0.019	92.550	0.687
0.01	1600	0.914	0.005	113.845	0.869
0.01	1750	0.927	0.033	135.668	4.128
0.01	1900	0.915	0.034	162.001	0.967
0.01	2050	0.961	0.018	191.184	1.094

Table A.27: Detailed data for Weighted - $\bar{p} = 0.010$.

Algorithm: Weighted					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	100	0.000	0.011	**0.478**	0.010
0.015	250	0.719	0.054	3.117	0.104
0.015	400	0.859	0.022	8.488	0.207
0.015	550	0.845	0.024	16.692	0.154
0.015	700	0.834	0.005	27.846	0.184
0.015	850	0.944	0.014	42.093	0.203
0.015	1000	0.842	0.078	57.351	2.621
0.015	1150	0.974	0.015	79.332	0.669
0.015	1300	0.963	0.004	101.882	1.039
0.015	1450	0.968	0.015	127.861	0.640
0.015	1600	0.921	0.001	157.770	1.419
0.015	1750	0.972	0.002	190.326	1.234
0.015	1900	0.975	0.019	226.502	2.242
0.015	2050	0.979	0.013	267.096	1.765

Table A.28: Detailed data for Weighted - $\bar{p} = 0.015$.

Algorithm: Weighted					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	100	-0.008	0.001	**0.609**	0.009
0.02	250	0.855	0.058	3.993	0.147
0.02	400	0.914	0.008	10.860	0.192
0.02	550	0.866	0.000	21.301	0.134
0.02	700	0.933	0.056	35.938	0.435
0.02	850	0.958	0.000	54.156	0.201
0.02	1000	0.978	0.008	75.955	0.365
0.02	1150	0.977	0.013	100.912	0.701
0.02	1300	0.969	0.000	130.336	0.644
0.02	1450	0.986	0.000	163.318	0.711
0.02	1600	0.994	0.006	201.110	0.660
0.02	1750	0.965	0.008	243.578	1.170
0.02	1900	0.977	0.014	284.639	9.150
0.02	2050	0.985	0.011	334.686	8.173

Table A.29: Detailed data for Weighted - $\bar{p} = 0.020$.

Algorithm: Weighted					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	100	-0.008	0.003	0.785	0.072
0.025	250	0.831	0.036	4.903	0.073
0.025	400	0.921	0.000	13.392	0.119
0.025	550	0.978	0.000	26.349	0.170
0.025	700	0.971	0.019	43.899	0.146
0.025	850	0.958	0.000	66.424	0.323
0.025	1000	0.981	0.007	93.197	0.779
0.025	1150	0.984	0.011	126.034	2.297
0.025	1300	0.971	0.009	158.694	4.411
0.025	1450	0.984	0.003	208.531	12.922
0.025	1600	0.996	0.005	248.929	1.784
0.025	1750	0.985	0.009	298.685	1.054
0.025	1900	0.986	0.003	353.253	0.857
0.025	2050	0.992	0.007	414.999	2.553

Table A.30: Detailed data for Weighted - $\bar{p} = 0.025$.

Algorithm: Adjusted					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	100	0.007	0.019	0.250	0.011
0.005	250	**0.073**	0.081	1.515	0.106
0.005	400	**0.101**	0.050	4.126	0.128
0.005	550	**0.390**	0.097	8.166	0.227
0.005	700	**0.476**	0.088	13.307	0.144
0.005	850	0.572	0.065	19.811	0.211
0.005	1000	0.660	0.060	27.752	0.455
0.005	1150	0.714	0.069	36.822	0.533
0.005	1300	0.775	0.035	47.224	0.389
0.005	1450	0.812	0.030	59.622	0.704
0.005	1600	0.816	0.039	69.951	2.699
0.005	1750	0.858	0.033	87.736	1.428
0.005	1900	0.880	0.038	103.821	1.513
0.005	2050	0.873	0.028	120.868	1.592

Table A.31: Detailed data for Adjusted - $\bar{p} = 0.005$.

Algorithm: Adjusted					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	100	**0.451**	0.264	0.450	0.086
0.01	250	**0.726**	0.094	2.567	0.125
0.01	400	0.846	0.043	6.949	0.237
0.01	550	0.882	0.038	13.607	0.543
0.01	700	0.932	0.031	22.660	0.878
0.01	850	0.959	0.022	32.948	0.642
0.01	1000	0.977	0.015	45.776	0.376
0.01	1150	0.981	0.005	61.422	0.223
0.01	1300	0.990	0.009	79.694	1.382
0.01	1450	0.992	0.005	99.186	0.332
0.01	1600	0.997	0.003	123.274	0.566
0.01	1750	0.995	0.004	145.920	5.346
0.01	1900	0.997	0.002	174.072	2.225
0.01	2050	0.996	0.004	204.708	2.838

Table A.32: Detailed data for Adjusted - $\bar{p} = 0.010$.

Algorithm: Adjusted					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	100	**0.817**	0.085	0.590	0.022
0.015	250	0.934	0.017	3.605	0.098
0.015	400	0.974	0.019	9.693	0.663
0.015	550	0.994	0.003	18.903	0.511
0.015	700	0.995	0.006	31.107	1.310
0.015	850	**1.000**	0.000	46.549	0.866
0.015	1000	0.998	0.002	63.299	1.289
0.015	1150	0.999	0.002	86.063	3.146
0.015	1300	0.998	0.003	112.717	4.443
0.015	1450	**1.000**	0.000	142.066	2.644
0.015	1600	**1.000**	0.000	173.557	0.964
0.015	1750	**1.000**	0.000	211.016	7.365
0.015	1900	**1.000**	0.001	246.004	6.587
0.015	2050	**1.000**	0.001	292.152	4.660

Table A.33: Detailed data for Adjusted - $\bar{p} = 0.015$.

Algorithm: Adjusted					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	100	**0.876**	0.072	0.751	0.024
0.02	250	0.971	0.021	4.617	0.112
0.02	400	**1.000**	0.000	12.280	0.335
0.02	550	0.997	0.004	24.881	0.996
0.02	700	**1.000**	0.000	40.840	1.521
0.02	850	0.999	0.002	61.425	1.489
0.02	1000	**1.000**	0.000	86.340	2.766
0.02	1150	**1.000**	0.000	112.522	2.519
0.02	1300	**1.000**	0.000	149.263	9.025
0.02	1450	**1.000**	0.000	192.212	8.035
0.02	1600	**1.000**	0.000	230.312	5.534
0.02	1750	**1.000**	0.000	278.259	10.126
0.02	1900	**1.000**	0.000	320.226	15.800
0.02	2050	**1.000**	0.000	395.460	29.602

Table A.34: Detailed data for Adjusted - $\bar{p} = 0.020$.

Algorithm: Adjusted					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	100	**0.961**	0.056	0.940	0.018
0.025	250	0.997	0.007	5.768	0.165
0.025	400	**1.000**	0.000	15.669	0.464
0.025	550	**1.000**	0.000	35.023	6.506
0.025	700	**1.000**	0.000	52.841	1.927
0.025	850	**1.000**	0.000	79.680	1.957
0.025	1000	**1.000**	0.000	109.416	3.005
0.025	1150	**1.000**	0.000	145.332	5.265
0.025	1300	**1.000**	0.000	183.403	7.780
0.025	1450	**1.000**	0.000	249.508	13.585
0.025	1600	**1.000**	0.000	295.687	7.637
0.025	1750	**1.000**	0.000	356.981	17.577
0.025	1900	**1.000**	0.000	413.325	23.371
0.025	2050	**1.000**	0.000	507.888	36.277

Table A.35: Detailed data for Adjusted - $\bar{p} = 0.025$.

Algorithm: SBMClone					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	100	0.000	0.000	0.718	0.094
0.005	250	0.000	0.000	2.264	0.393
0.005	400	0.000	0.000	3.941	0.480
0.005	550	0.000	0.000	**6.400**	0.864
0.005	700	0.000	0.000	12.776	1.333
0.005	850	**0.895**	0.020	60.408	16.829
0.005	1000	**0.949**	0.033	70.458	16.362
0.005	1150	**0.989**	0.007	77.508	23.301
0.005	1300	**0.996**	0.003	94.541	36.944
0.005	1450	**0.998**	0.002	110.245	42.147
0.005	1600	**0.997**	0.003	84.087	13.344
0.005	1750	**0.999**	0.001	115.263	24.009
0.005	1900	**0.999**	0.002	98.092	2.298
0.005	2050	**1.000**	0.000	**105.658**	2.825

Table A.36: Detailed data for SBMClone - $\bar{p} = 0.005$.

Algorithm: SBMClone					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	100	0.000	0.000	1.947	0.133
0.01	250	0.000	0.000	7.180	1.242
0.01	400	**0.990**	0.007	27.981	5.484
0.01	550	**1.000**	0.000	43.418	9.216
0.01	700	**1.000**	0.000	72.831	20.711
0.01	850	**1.000**	0.000	93.769	28.329
0.01	1000	**1.000**	0.000	141.399	73.596
0.01	1150	**1.000**	0.000	309.602	432.737
0.01	1300	**1.000**	0.000	240.801	251.976
0.01	1450	**1.000**	0.000	208.034	107.804
0.01	1600	**1.000**	0.000	181.330	38.589
0.01	1750	**1.000**	0.000	283.503	178.799
0.01	1900	**1.000**	0.000	206.551	43.694
0.01	2050	**1.000**	0.000	408.131	269.555

Table A.37: Detailed data for SBMClone - $\bar{p} = 0.010$.

Algorithm: SBMClone					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	100	0.000	0.000	3.180	0.300
0.015	250	**1.000**	0.000	31.190	7.558
0.015	400	**1.000**	0.000	56.273	20.280
0.015	550	**1.000**	0.000	63.256	15.570
0.015	700	**0.999**	0.001	95.092	17.628
0.015	850	**1.000**	0.000	99.387	3.777
0.015	1000	**1.000**	0.000	194.618	133.915
0.015	1150	**1.000**	0.000	186.053	44.904
0.015	1300	**1.000**	0.000	162.393	2.661
0.015	1450	**1.000**	0.000	354.075	154.274
0.015	1600	**1.000**	0.000	296.152	164.053
0.015	1750	**1.000**	0.000	258.653	9.366
0.015	1900	**1.000**	0.000	240.250	8.670
0.015	2050	**1.000**	0.000	289.842	70.686

Table A.38: Detailed data for SBMClone - $\bar{p} = 0.015$.

Algorithm: SBMClone					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	100	0.000	0.000	4.908	0.559
0.02	250	**1.000**	0.000	37.081	14.479
0.02	400	**1.000**	0.000	75.219	36.525
0.02	550	**1.000**	0.000	102.040	27.702
0.02	700	**1.000**	0.000	117.779	33.044
0.02	850	**1.000**	0.000	128.656	7.041
0.02	1000	**1.000**	0.000	230.722	156.593
0.02	1150	**1.000**	0.000	239.490	31.729
0.02	1300	**1.000**	0.000	303.356	150.458
0.02	1450	**1.000**	0.000	231.832	7.380
0.02	1600	**1.000**	0.000	316.290	59.843
0.02	1750	**1.000**	0.000	277.803	6.395
0.02	1900	**1.000**	0.000	306.523	10.878
0.02	2050	**1.000**	0.000	618.983	126.910

Table A.39: Detailed data for SBMClone - $\bar{p} = 0.020$.

Algorithm: SBMClone					
$\bar{p}$	m	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	100	0.200	0.447	9.282	3.490
0.025	250	**1.000**	0.000	35.044	4.837
0.025	400	**1.000**	0.000	87.969	18.157
0.025	550	**1.000**	0.000	106.183	9.505
0.025	700	**1.000**	0.000	130.509	18.614
0.025	850	**1.000**	0.000	217.240	27.772
0.025	1000	**1.000**	0.000	185.067	9.963
0.025	1150	**1.000**	0.000	229.101	23.969
0.025	1300	**1.000**	0.000	242.168	10.603
0.025	1450	**1.000**	0.000	309.224	74.700
0.025	1600	**1.000**	0.000	342.856	18.475
0.025	1750	**1.000**	0.000	518.340	128.882
0.025	1900	**1.000**	0.000	621.997	283.567
0.025	2050	**1.000**	0.000	413.310	22.548

Table A.40: Detailed data for SBMClone - $\bar{p} = 0.025$.

A.3 Number of mutations

Algorithm: Naive					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	100	0.000	0.000	138.955	1.567
0.005	250	0.000	0.000	161.981	1.340
0.005	400	-0.000	0.000	172.313	2.731
0.005	550	-0.000	0.000	182.600	1.508
0.005	700	0.000	0.000	190.998	1.974
0.005	850	0.001	0.001	198.365	2.439
0.005	1000	0.000	0.000	205.784	1.831
0.005	1150	0.001	0.001	210.065	3.748
0.005	1300	0.002	0.002	220.759	3.323
0.005	1450	0.001	0.001	229.724	2.298
0.005	1600	0.001	0.001	234.516	4.679
0.005	1750	0.005	0.004	241.434	5.688
0.005	1900	0.001	0.001	246.850	3.976
0.005	2050	0.003	0.003	256.416	3.806

Table A.41: Detailed data for Naive - $\bar{p} = 0.005$.

Algorithm: Naive					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	100	**0.001**	0.001	154.828	1.782
0.01	250	0.000	0.001	178.236	2.364
0.01	400	0.000	0.000	194.322	1.657
0.01	550	0.001	0.001	211.019	1.661
0.01	700	0.002	0.003	226.807	3.736
0.01	850	0.014	0.013	244.472	2.248
0.01	1000	0.086	0.080	255.164	8.803
0.01	1150	0.152	0.106	272.386	2.175
0.01	1300	0.128	0.089	285.740	4.412
0.01	1450	0.261	0.077	303.197	4.232
0.01	1600	0.349	0.073	315.840	2.338
0.01	1750	0.349	0.026	332.929	2.548
0.01	1900	0.401	0.114	347.036	3.323
0.01	2050	0.463	0.027	363.420	3.257

Table A.42: Detailed data for Naive - $\bar{p} = 0.010$.

Algorithm: Naive					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	100	**0.000**	0.000	164.287	2.537
0.015	250	-0.000	0.000	189.167	1.575
0.015	400	0.001	0.002	215.828	1.178
0.015	550	0.003	0.004	239.282	2.721
0.015	700	0.108	0.098	264.204	2.027
0.015	850	0.235	0.144	284.243	2.746
0.015	1000	0.418	0.098	307.347	3.416
0.015	1150	0.412	0.108	327.558	7.943
0.015	1300	0.545	0.056	350.149	2.658
0.015	1450	0.584	0.047	377.435	3.151
0.015	1600	0.568	0.111	399.895	3.837
0.015	1750	0.653	0.014	426.787	2.659
0.015	1900	0.688	0.024	447.216	5.207
0.015	2050	0.706	0.008	461.843	1.580

Table A.43: Detailed data for Naive - $\bar{p} = 0.015$.

Algorithm: Naive					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	100	0.000	0.000	171.398	3.559
0.02	250	0.000	0.001	198.329	1.710
0.02	400	0.103	0.101	232.595	1.757
0.02	550	0.322	0.090	262.992	1.999
0.02	700	0.470	0.128	292.948	2.587
0.02	850	0.454	0.178	317.971	3.841
0.02	1000	0.658	0.024	347.823	0.739
0.02	1150	0.680	0.034	382.098	1.524
0.02	1300	0.735	0.017	411.857	0.960
0.02	1450	0.745	0.016	442.487	3.652
0.02	1600	0.763	0.029	472.265	2.318
0.02	1750	0.797	0.020	506.991	4.446
0.02	1900	0.796	0.035	530.643	5.289
0.02	2050	0.809	0.043	561.956	4.421

Table A.44: Detailed data for Naive - $\bar{p} = 0.020$.

Algorithm: Naive					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	100	0.001	0.001	175.210	3.345
0.025	250	0.000	0.000	211.585	3.328
0.025	400	0.287	0.180	250.216	3.400
0.025	550	0.569	0.054	291.061	2.512
0.025	700	0.512	0.289	325.126	3.361
0.025	850	0.708	0.042	362.841	3.594
0.025	1000	0.752	0.012	400.404	6.247
0.025	1150	0.787	0.019	439.738	2.101
0.025	1300	0.816	0.023	481.736	3.076
0.025	1450	0.843	0.026	512.095	13.019
0.025	1600	0.863	0.014	559.597	14.788
0.025	1750	0.872	0.019	588.583	3.352
0.025	1900	0.883	0.014	625.436	3.224
0.025	2050	0.888	0.009	659.725	2.636

Table A.45: Detailed data for Naive - $\bar{p} = 0.025$.

Algorithm: Weighted					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	100	-0.000	0.000	38817.553	1469.551
0.005	250	**0.000**	0.000	8684.403	495.209
0.005	400	-0.000	0.000	2216.916	49.338
0.005	550	**-0.000**	0.000	677.228	18.108
0.005	700	0.000	0.000	331.460	7.466
0.005	850	0.000	0.000	262.798	2.967
0.005	1000	0.000	0.000	257.078	3.796
0.005	1150	0.000	0.000	256.467	6.300
0.005	1300	0.000	0.000	265.168	4.208
0.005	1450	0.000	0.000	271.122	1.472
0.005	1600	0.000	0.000	276.074	2.803
0.005	1750	0.000	0.000	284.392	2.300
0.005	1900	0.000	0.000	291.337	3.219
0.005	2050	0.000	0.000	300.129	2.727

Table A.46: Detailed data for Weighted - $\bar{p} = 0.005$.

Algorithm: Weighted					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	100	-0.000	0.000	14407.092	807.978
0.01	250	-0.000	0.000	900.981	80.053
0.01	400	-0.000	0.000	270.060	4.997
0.01	550	0.000	0.000	256.775	1.458
0.01	700	0.000	0.000	270.154	1.043
0.01	850	0.000	0.000	285.131	2.159
0.01	1000	0.000	0.000	295.412	9.352
0.01	1150	0.165	0.368	314.939	3.083
0.01	1300	0.503	0.460	328.547	2.653
0.01	1450	0.684	0.382	346.489	2.950
0.01	1600	0.697	0.390	365.102	1.625
0.01	1750	0.718	0.401	380.433	2.618
0.01	1900	0.899	0.025	396.612	3.318
0.01	2050	0.933	0.007	416.592	2.178

Table A.47: Detailed data for Weighted - $\bar{p} = 0.010$.

Algorithm: Weighted					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	100	-0.000	0.000	5346.111	465.720
0.015	250	0.000	0.000	278.840	6.522
0.015	400	0.000	0.000	258.981	3.229
0.015	550	0.000	0.000	280.870	4.691
0.015	700	0.148	0.331	307.279	2.611
0.015	850	0.515	0.471	329.403	2.317
0.015	1000	0.535	0.489	352.213	2.404
0.015	1150	0.915	0.009	378.154	2.363
0.015	1300	0.927	0.022	403.426	1.961
0.015	1450	0.950	0.004	430.434	3.145
0.015	1600	0.955	0.008	457.127	2.434
0.015	1750	0.934	0.024	484.795	1.106
0.015	1900	0.969	0.008	508.147	4.530
0.015	2050	0.965	0.007	525.199	2.806

Table A.48: Detailed data for Weighted - $\bar{p} = 0.015$.

Algorithm: Weighted					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	100	-0.000	0.000	2141.701	181.561
0.02	250	0.000	0.000	246.692	4.153
0.02	400	0.000	0.000	278.290	1.016
0.02	550	0.000	0.000	308.527	3.147
0.02	700	0.725	0.406	338.327	0.550
0.02	850	0.925	0.015	368.203	3.798
0.02	1000	0.950	0.011	404.491	2.678
0.02	1150	0.960	0.013	439.689	2.613
0.02	1300	0.971	0.006	476.033	2.366
0.02	1450	0.974	0.004	506.685	4.451
0.02	1600	0.979	0.006	543.649	3.245
0.02	1750	0.977	0.005	574.351	2.282
0.02	1900	0.984	0.008	603.695	11.413
0.02	2050	0.982	0.001	642.151	3.002

Table A.49: Detailed data for Weighted - $\bar{p} = 0.020$.

Algorithm: Weighted					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	100	-0.000	0.000	888.164	71.370
0.025	250	0.000	0.000	257.762	2.727
0.025	400	0.000	0.000	295.080	2.646
0.025	550	0.731	0.409	337.974	1.706
0.025	700	0.951	0.008	377.811	1.495
0.025	850	0.962	0.005	419.745	1.859
0.025	1000	0.975	0.005	464.826	3.241
0.025	1150	0.982	0.005	508.557	3.075
0.025	1300	0.982	0.005	550.752	4.867
0.025	1450	0.984	0.002	592.419	3.984
0.025	1600	0.984	0.008	633.153	2.211
0.025	1750	0.985	0.005	675.740	0.353
0.025	1900	0.990	0.008	713.087	3.658
0.025	2050	0.987	0.005	753.064	2.563

Table A.50: Detailed data for Weighted - $\bar{p} = 0.025$.

Algorithm: Adjusted					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	100	**0.001**	0.001	39574.937	2396.776
0.005	250	-0.000	0.000	8924.613	574.722
0.005	400	-0.000	0.000	2322.590	57.214
0.005	550	**-0.000**	0.000	753.464	22.333
0.005	700	0.000	0.000	396.994	10.046
0.005	850	0.000	0.000	320.125	4.873
0.005	1000	0.000	0.000	311.283	2.817
0.005	1150	0.000	0.000	309.054	8.867
0.005	1300	0.000	0.000	323.550	2.646
0.005	1450	0.000	0.000	329.897	0.991
0.005	1600	0.000	0.000	338.674	4.435
0.005	1750	0.000	0.000	344.853	7.742
0.005	1900	0.000	0.000	357.208	3.028
0.005	2050	0.000	0.000	365.271	2.186

Table A.51: Detailed data for Adjusted - $\bar{p} = 0.005$.

Algorithm: Adjusted					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	100	-0.000	0.000	14748.854	781.089
0.01	250	-0.000	0.000	981.519	79.793
0.01	400	-0.000	0.000	328.184	6.620
0.01	550	0.000	0.000	310.024	4.840
0.01	700	0.000	0.000	325.611	4.278
0.01	850	0.000	0.000	340.377	4.334
0.01	1000	0.000	0.000	359.806	5.164
0.01	1150	0.167	0.373	335.654	32.903
0.01	1300	0.525	0.480	344.299	30.561
0.01	1450	0.738	0.413	351.400	2.367
0.01	1600	0.752	0.420	369.597	2.343
0.01	1750	0.768	0.430	389.461	4.670
0.01	1900	0.966	0.007	403.693	3.260
0.01	2050	0.969	0.008	426.684	3.671

Table A.52: Detailed data for Adjusted - $\bar{p} = 0.010$.

Algorithm: Adjusted					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	100	-0.000	0.000	5484.290	483.011
0.015	250	0.000	0.000	337.970	8.790
0.015	400	0.000	0.000	311.331	3.756
0.015	550	0.000	0.000	337.783	4.083
0.015	700	0.167	0.372	342.709	36.975
0.015	850	0.553	0.505	353.066	37.048
0.015	1000	0.570	0.520	356.499	3.948
0.015	1150	0.963	0.008	380.732	8.276
0.015	1300	0.979	0.002	413.577	4.335
0.015	1450	0.988	0.004	445.461	4.128
0.015	1600	0.986	0.005	471.050	3.742
0.015	1750	0.992	0.004	503.841	4.708
0.015	1900	0.994	0.003	1142.884	1357.854
0.015	2050	0.994	0.003	1135.188	1181.348

Table A.53: Detailed data for Adjusted - $\bar{p} = 0.015$.

Algorithm: Adjusted					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	100	-0.000	0.000	2257.943	164.179
0.02	250	0.000	0.000	299.678	3.297
0.02	400	0.000	0.000	332.459	4.216
0.02	550	0.000	0.000	367.364	4.303
0.02	700	0.753	0.421	340.689	6.157
0.02	850	0.963	0.012	375.860	4.729
0.02	1000	0.982	0.006	413.101	2.682
0.02	1150	0.992	0.004	446.064	2.014
0.02	1300	0.995	0.002	487.406	5.783
0.02	1450	0.997	0.001	524.391	3.460
0.02	1600	0.997	0.002	565.623	4.955
0.02	1750	0.998	0.002	601.980	5.698
0.02	1900	0.999	0.001	645.355	8.878
0.02	2050	0.999	0.001	679.424	2.913

Table A.54: Detailed data for Adjusted - $\bar{p} = 0.020$.

Algorithm: Adjusted					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	100	-0.000	0.000	964.971	73.298
0.025	250	0.000	0.000	312.386	4.399
0.025	400	0.000	0.000	341.744	27.840
0.025	550	0.751	0.420	338.154	3.867
0.025	700	0.980	0.008	381.024	4.188
0.025	850	0.987	0.009	420.058	13.837
0.025	1000	0.994	0.002	472.628	5.481
0.025	1150	0.996	0.002	517.931	10.675
0.025	1300	0.998	0.002	562.874	10.481
0.025	1450	0.999	0.001	615.884	10.898
0.025	1600	**1.000**	0.000	659.468	10.077
0.025	1750	**1.000**	0.000	706.829	6.267
0.025	1900	**1.000**	0.000	758.854	13.968
0.025	2050	**1.000**	0.000	805.297	3.129

Table A.55: Detailed data for Adjusted - $\bar{p} = 0.025$.

Algorithm: SBMClone					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	100	0.000	0.000	**0.604**	0.044
0.005	250	-0.000	0.000	**2.035**	0.426
0.005	400	**0.000**	0.000	**3.533**	0.396
0.005	550	-0.000	0.000	**5.137**	1.226
0.005	700	**0.104**	0.059	**19.439**	7.364
0.005	850	**0.301**	0.158	**27.133**	2.010
0.005	1000	**0.414**	0.141	**46.394**	11.304
0.005	1150	**0.528**	0.115	**55.668**	19.101
0.005	1300	**0.800**	0.014	**77.897**	9.769
0.005	1450	**0.848**	0.006	**97.727**	32.584
0.005	1600	**0.885**	0.006	**91.241**	32.298
0.005	1750	**0.912**	0.005	**79.157**	20.489
0.005	1900	**0.931**	0.006	**105.776**	33.705
0.005	2050	**0.947**	0.007	**108.116**	38.367

Table A.56: Detailed data for SBMClone - $\bar{p} = 0.005$.

Algorithm: SBMClone					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	100	0.000	0.000	**1.600**	0.061
0.01	250	**0.041**	0.002	**11.320**	2.754
0.01	400	**0.276**	0.155	**28.479**	12.933
0.01	550	**0.722**	0.012	**40.781**	19.455
0.01	700	**0.841**	0.008	**54.812**	18.102
0.01	850	**0.897**	0.006	**56.911**	12.662
0.01	1000	**0.945**	0.007	**129.478**	95.200
0.01	1150	**0.972**	0.006	**76.318**	6.095
0.01	1300	**0.982**	0.003	**147.954**	60.564
0.01	1450	**0.987**	0.001	**199.368**	69.636
0.01	1600	**0.992**	0.002	**149.296**	41.964
0.01	1750	**0.997**	0.002	**143.676**	23.503
0.01	1900	**0.996**	0.002	**153.205**	27.708
0.01	2050	**0.999**	0.002	**158.646**	7.319

Table A.57: Detailed data for SBMClone - $\bar{p} = 0.010$.

Algorithm: SBMClone					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	100	-0.000	0.000	**2.591**	0.295
0.015	250	**0.248**	0.164	**23.364**	10.721
0.015	400	**0.772**	0.006	**30.409**	7.749
0.015	550	**0.895**	0.007	**67.590**	20.450
0.015	700	**0.952**	0.008	**87.644**	9.675
0.015	850	**0.977**	0.004	**89.222**	4.157
0.015	1000	**0.987**	0.004	**118.640**	25.648
0.015	1150	**0.995**	0.002	**119.167**	11.518
0.015	1300	**0.999**	0.001	**129.137**	6.755
0.015	1450	**0.999**	0.001	**165.313**	10.606
0.015	1600	**0.999**	0.001	**176.746**	5.237
0.015	1750	**1.000**	0.000	**259.672**	74.214
0.015	1900	**1.000**	0.000	**360.267**	80.736
0.015	2050	**1.000**	0.000	**235.549**	10.253

Table A.58: Detailed data for SBMClone - $\bar{p} = 0.015$.

Algorithm: SBMClone					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	100	**0.019**	0.001	**7.254**	1.456
0.02	250	**0.561**	0.180	**41.944**	14.003
0.02	400	**0.886**	0.015	**75.569**	68.599
0.02	550	**0.963**	0.005	**76.316**	21.751
0.02	700	**0.985**	0.003	**156.945**	115.084
0.02	850	**0.994**	0.003	**126.702**	33.724
0.02	1000	**0.998**	0.002	**139.484**	9.690
0.02	1150	**0.999**	0.001	**148.334**	5.453
0.02	1300	**1.000**	0.000	**207.890**	11.639
0.02	1450	**1.000**	0.000	**313.572**	183.822
0.02	1600	**1.000**	0.000	**220.414**	10.300
0.02	1750	**1.000**	0.000	**261.225**	67.391
0.02	1900	**1.000**	0.000	**259.965**	21.645
0.02	2050	**1.000**	0.000	**401.673**	262.314

Table A.59: Detailed data for SBMClone - $\bar{p} = 0.020$.

Algorithm: SBMClone					
$\bar{p}$	n	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	100	**0.057**	0.024	**11.252**	5.134
0.025	250	**0.795**	0.010	**33.514**	8.998
0.025	400	**0.942**	0.011	**84.447**	47.143
0.025	550	**0.987**	0.003	**94.783**	18.973
0.025	700	**0.994**	0.003	**113.758**	9.812
0.025	850	**0.999**	0.001	**198.235**	155.992
0.025	1000	**0.999**	0.001	**177.918**	7.729
0.025	1150	**1.000**	0.001	**221.114**	34.528
0.025	1300	**1.000**	0.000	**214.321**	7.425
0.025	1450	**1.000**	0.000	**230.813**	17.524
0.025	1600	**1.000**	0.000	**252.286**	12.272
0.025	1750	**1.000**	0.000	**285.320**	14.783
0.025	1900	**1.000**	0.000	**349.107**	92.298
0.025	2050	**1.000**	0.000	**551.833**	78.517

Table A.60: Detailed data for SBMClone - $\bar{p} = 0.025$.

A.4 Number of both cells and mutations

Algorithm: Naive					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	100	-0.008	0.002	**0.048**	0.007
0.005	250	0.004	0.005	0.385	0.059
0.005	400	-0.002	0.001	1.231	0.055
0.005	550	**0.002**	0.004	2.646	0.102
0.005	700	**0.002**	0.002	4.792	0.083
0.005	850	-0.000	0.001	7.597	0.082
0.005	1000	**0.007**	0.007	11.130	0.096
0.005	1150	**0.005**	0.008	15.361	0.224
0.005	1300	**0.003**	0.005	20.328	0.256
0.005	1450	**0.002**	0.002	26.847	0.318
0.005	1600	0.009	0.012	34.358	0.362
0.005	1750	0.018	0.012	43.107	0.512
0.005	1900	0.019	0.023	52.091	0.893
0.005	2050	0.039	0.030	62.769	0.531

Table A.61: Detailed data for Naive - $\bar{p} = 0.005$.

Algorithm: Naive					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	100	-0.002	0.008	0.076	0.048
0.01	250	0.005	0.010	0.479	0.062
0.01	400	**0.011**	0.017	1.483	0.083
0.01	550	**0.001**	0.004	3.178	0.101
0.01	700	**0.006**	0.005	5.839	0.071
0.01	850	0.046	0.060	**9.402**	0.230
0.01	1000	0.024	0.045	**14.009**	0.137
0.01	1150	0.146	0.038	**20.052**	0.299
0.01	1300	0.151	0.052	**27.346**	0.390
0.01	1450	0.236	0.037	**36.026**	0.419
0.01	1600	0.267	0.082	**46.779**	0.527
0.01	1750	0.335	0.036	**59.317**	0.664
0.01	1900	0.313	0.030	**73.841**	0.767
0.01	2050	0.333	0.021	89.371	1.166

Table A.62: Detailed data for Naive - $\bar{p} = 0.010$.

Algorithm: Naive					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	100	0.003	0.020	0.079	0.043
0.015	250	**0.019**	0.029	0.532	0.063
0.015	400	**0.008**	0.011	1.636	0.060
0.015	550	0.075	0.099	**3.644**	0.078
0.015	700	0.216	0.065	**6.813**	0.115
0.015	850	0.205	0.061	**11.039**	0.149
0.015	1000	0.326	0.051	**16.793**	0.173
0.015	1150	0.365	0.035	**23.688**	0.897
0.015	1300	0.430	0.075	**32.321**	1.286
0.015	1450	0.422	0.097	**45.269**	0.551
0.015	1600	0.524	0.059	**59.503**	0.306
0.015	1750	0.557	0.048	**76.614**	0.627
0.015	1900	0.553	0.072	**95.682**	0.926
0.015	2050	0.628	0.033	**117.173**	2.194

Table A.63: Detailed data for Naive - $\bar{p} = 0.015$.

Algorithm: Naive					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	100	**0.012**	0.028	**0.066**	0.007
0.02	250	**0.043**	0.058	0.548	0.056
0.02	400	0.038	0.044	**1.839**	0.069
0.02	550	0.225	0.072	**4.148**	0.060
0.02	700	0.331	0.042	**7.727**	0.064
0.02	850	0.388	0.070	**12.806**	0.230
0.02	1000	0.479	0.022	**19.680**	0.369
0.02	1150	0.534	0.042	**29.955**	2.544
0.02	1300	0.607	0.049	**40.611**	0.423
0.02	1450	0.630	0.049	**54.935**	0.360
0.02	1600	0.704	0.046	**70.066**	2.211
0.02	1750	0.719	0.035	**93.094**	0.853
0.02	1900	0.738	0.016	**115.694**	0.584
0.02	2050	0.767	0.009	**143.034**	1.302

Table A.64: Detailed data for Naive - $\bar{p} = 0.020$.

Algorithm: Naive					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	100	**0.025**	0.040	**0.080**	0.039
0.025	250	**0.047**	0.070	**0.594**	0.053
0.025	400	0.102	0.084	**1.958**	0.059
0.025	550	0.366	0.043	**4.533**	0.091
0.025	700	0.511	0.063	**8.623**	0.070
0.025	850	0.543	0.113	**14.520**	0.105
0.025	1000	0.648	0.073	**22.827**	0.182
0.025	1150	0.684	0.032	**33.538**	0.417
0.025	1300	0.704	0.042	**47.277**	0.220
0.025	1450	0.771	0.037	**63.945**	0.315
0.025	1600	0.752	0.041	**84.984**	1.377
0.025	1750	0.824	0.022	**107.875**	0.584
0.025	1900	0.855	0.031	**136.384**	0.519
0.025	2050	0.855	0.023	**168.202**	1.330

Table A.65: Detailed data for Naive - $\bar{p} = 0.025$.

Algorithm: Weighted					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	100	-0.000	0.019	0.822	0.081
0.005	250	0.002	0.005	7.637	1.168
0.005	400	**0.002**	0.006	8.453	1.588
0.005	550	-0.000	0.001	6.202	0.549
0.005	700	-0.000	0.001	6.890	0.256
0.005	850	0.000	0.001	9.145	0.217
0.005	1000	-0.000	0.000	12.775	0.269
0.005	1150	-0.000	0.000	17.565	0.267
0.005	1300	-0.000	0.000	23.433	0.150
0.005	1450	-0.000	0.000	30.456	0.210
0.005	1600	-0.000	0.000	39.314	0.505
0.005	1750	0.000	0.000	49.129	0.526
0.005	1900	0.000	0.000	59.455	0.897
0.005	2050	0.000	0.000	71.499	0.826

Table A.66: Detailed data for Weighted - $\bar{p} = 0.005$.

Algorithm: Weighted					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	100	-0.007	0.005	0.695	0.111
0.01	250	**0.012**	0.012	1.045	0.139
0.01	400	-0.000	0.001	1.682	0.079
0.01	550	0.000	0.000	3.518	0.090
0.01	700	-0.000	0.000	6.413	0.032
0.01	850	0.000	0.000	10.353	0.060
0.01	1000	0.065	0.146	15.400	0.152
0.01	1150	0.312	0.306	21.994	0.283
0.01	1300	0.446	0.255	29.987	0.274
0.01	1450	0.548	0.310	40.014	0.636
0.01	1600	0.744	0.045	51.944	0.409
0.01	1750	0.801	0.052	65.735	0.742
0.01	1900	0.822	0.014	81.900	0.514
0.01	2050	0.784	0.057	100.918	3.212

Table A.67: Detailed data for Weighted - $\bar{p} = 0.010$.

Algorithm: Weighted					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	100	**0.015**	0.035	0.401	0.132
0.015	250	0.005	0.005	0.628	0.094
0.015	400	-0.000	0.000	1.770	0.078
0.015	550	0.058	0.129	3.964	0.036
0.015	700	0.063	0.141	7.388	0.056
0.015	850	0.589	0.018	12.133	0.204
0.015	1000	0.621	0.350	18.406	0.176
0.015	1150	0.814	0.061	25.905	1.122
0.015	1300	0.843	0.030	35.510	1.725
0.015	1450	0.856	0.028	49.993	0.628
0.015	1600	0.898	0.023	65.521	0.569
0.015	1750	0.899	0.029	83.758	0.451
0.015	1900	0.926	0.013	105.381	1.332
0.015	2050	0.937	0.037	129.325	1.193

Table A.68: Detailed data for Weighted - $\bar{p} = 0.015$.

Algorithm: Weighted					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	100	-0.008	0.003	0.194	0.045
0.02	250	0.001	0.002	0.615	0.049
0.02	400	0.038	0.086	1.933	0.057
0.02	550	0.439	0.246	4.467	0.041
0.02	700	0.719	0.029	8.236	0.077
0.02	850	0.802	0.060	13.811	0.237
0.02	1000	0.861	0.036	21.370	0.290
0.02	1150	0.887	0.057	31.487	0.244
0.02	1300	0.888	0.057	43.985	0.363
0.02	1450	0.942	0.019	59.896	0.366
0.02	1600	0.944	0.032	76.421	2.061
0.02	1750	0.946	0.020	101.668	0.529
0.02	1900	0.953	0.015	128.207	1.134
0.02	2050	0.952	0.023	159.482	0.754

Table A.69: Detailed data for Weighted - $\bar{p} = 0.020$.

Algorithm: Weighted					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	100	-0.002	0.011	0.095	0.012
0.025	250	-0.000	0.000	0.618	0.052
0.025	400	0.231	0.213	2.056	0.052
0.025	550	0.729	0.028	4.874	0.043
0.025	700	0.822	0.065	9.249	0.127
0.025	850	0.897	0.025	15.697	0.109
0.025	1000	0.925	0.023	24.627	0.206
0.025	1150	0.935	0.027	36.418	0.135
0.025	1300	0.953	0.010	51.433	0.575
0.025	1450	0.963	0.015	69.877	0.760
0.025	1600	0.943	0.032	92.425	0.269
0.025	1750	0.984	0.012	118.680	1.257
0.025	1900	0.972	0.017	149.496	0.801
0.025	2050	0.988	0.006	186.643	0.706

Table A.70: Detailed data for Weighted - $\bar{p} = 0.025$.

Algorithm: Adjusted					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	100	0.001	0.008	0.926	0.132
0.005	250	-0.001	0.004	8.036	1.254
0.005	400	-0.000	0.002	9.061	1.730
0.005	550	0.000	0.001	6.690	0.467
0.005	700	-0.000	0.000	7.886	0.293
0.005	850	-0.000	0.000	10.831	0.211
0.005	1000	-0.000	0.000	15.172	0.242
0.005	1150	0.000	0.000	20.433	0.374
0.005	1300	0.000	0.000	27.928	0.354
0.005	1450	0.000	0.000	36.047	0.332
0.005	1600	0.000	0.000	46.416	0.384
0.005	1750	0.000	0.000	57.447	0.610
0.005	1900	0.000	0.000	69.315	5.131
0.005	2050	0.000	0.000	86.734	0.673

Table A.71: Detailed data for Adjusted - $\bar{p} = 0.005$.

Algorithm: Adjusted					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	100	**-0.001**	0.008	0.719	0.094
0.01	250	0.007	0.008	1.136	0.182
0.01	400	-0.000	0.000	2.042	0.120
0.01	550	0.000	0.000	4.163	0.086
0.01	700	-0.000	0.000	7.577	0.124
0.01	850	0.000	0.000	11.732	0.753
0.01	1000	0.136	0.304	16.504	1.371
0.01	1150	0.418	0.383	22.182	0.407
0.01	1300	0.626	0.351	30.540	0.361
0.01	1450	0.669	0.375	40.875	0.367
0.01	1600	0.862	0.021	53.596	0.354
0.01	1750	0.913	0.022	67.644	0.804
0.01	1900	0.927	0.015	83.933	1.332
0.01	2050	0.945	0.010	103.008	0.661

Table A.72: Detailed data for Adjusted - $\bar{p} = 0.010$.

Algorithm: Adjusted					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	100	0.004	0.015	0.436	0.132
0.015	250	0.001	0.001	0.738	0.043
0.015	400	0.000	0.000	2.115	0.074
0.015	550	0.111	0.248	4.485	0.241
0.015	700	0.115	0.256	7.785	0.518
0.015	850	0.805	0.018	12.558	0.171
0.015	1000	0.713	0.399	19.042	0.259
0.015	1150	0.926	0.009	26.814	1.064
0.015	1300	0.949	0.011	36.707	1.231
0.015	1450	0.961	0.009	51.432	0.484
0.015	1600	0.972	0.006	67.897	0.266
0.015	1750	0.975	0.020	86.878	0.507
0.015	1900	0.987	0.003	109.729	0.708
0.015	2050	0.991	0.005	136.021	0.739

Table A.73: Detailed data for Adjusted - $\bar{p} = 0.015$.

Algorithm: Adjusted					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	100	0.010	0.028	0.198	0.033
0.02	250	0.000	0.000	0.736	0.055
0.02	400	0.068	0.152	2.138	0.215
0.02	550	0.555	0.318	4.616	0.096
0.02	700	0.852	0.036	8.541	0.112
0.02	850	0.912	0.031	14.526	0.152
0.02	1000	0.959	0.022	22.450	0.260
0.02	1150	0.968	0.013	33.162	0.288
0.02	1300	0.985	0.008	46.267	0.806
0.02	1450	0.982	0.005	63.059	0.579
0.02	1600	0.998	0.001	81.517	2.062
0.02	1750	0.997	0.004	107.519	1.010
0.02	1900	0.999	0.002	135.241	1.035
0.02	2050	0.997	0.002	168.759	0.876

Table A.74: Detailed data for Adjusted - $\bar{p} = 0.020$.

Algorithm: Adjusted					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	100	-0.002	0.005	0.119	0.016
0.025	250	0.000	0.000	0.731	0.043
0.025	400	0.382	0.352	2.162	0.068
0.025	550	0.881	0.060	5.047	0.074
0.025	700	0.942	0.025	9.685	0.066
0.025	850	0.968	0.023	16.497	0.230
0.025	1000	0.982	0.013	26.001	0.265
0.025	1150	0.992	0.008	38.354	0.487
0.025	1300	0.998	0.004	54.634	1.551
0.025	1450	0.997	0.003	74.724	0.877
0.025	1600	0.997	0.003	98.465	1.210
0.025	1750	**1.000**	0.001	128.074	4.272
0.025	1900	**1.000**	0.000	161.330	2.143
0.025	2050	0.999	0.001	204.010	5.984

Table A.75: Detailed data for Adjusted - $\bar{p} = 0.025$.

Algorithm: SBMClone					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.005	100	**0.002**	0.007	0.049	0.003
0.005	250	**0.004**	0.006	**0.126**	0.002
0.005	400	-0.001	0.003	**0.260**	0.029
0.005	550	-0.000	0.000	**0.586**	0.025
0.005	700	-0.000	0.001	**0.993**	0.071
0.005	850	**0.000**	0.000	**1.993**	0.180
0.005	1000	-0.000	0.000	**2.852**	0.314
0.005	1150	-0.000	0.000	**5.192**	1.687
0.005	1300	-0.000	0.000	**8.582**	5.731
0.005	1450	-0.000	0.000	**7.894**	0.506
0.005	1600	**0.481**	0.439	**24.426**	9.265
0.005	1750	**0.854**	0.033	**36.293**	23.976
0.005	1900	**0.904**	0.008	**51.063**	19.563
0.005	2050	**0.931**	0.008	**61.621**	17.909

Table A.76: Detailed data for SBMClone - $\bar{p} = 0.005$.

Algorithm: SBMClone					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.01	100	-0.001	0.010	**0.066**	0.004
0.01	250	0.008	0.011	**0.218**	0.027
0.01	400	-0.000	0.001	**0.700**	0.065
0.01	550	0.000	0.000	**1.595**	0.169
0.01	700	-0.000	0.000	**3.058**	0.555
0.01	850	**0.702**	0.394	10.719	4.107
0.01	1000	**0.918**	0.018	21.263	7.334
0.01	1150	**0.959**	0.012	23.303	4.416
0.01	1300	**0.976**	0.007	31.828	7.850
0.01	1450	**0.987**	0.007	40.686	11.126
0.01	1600	**0.993**	0.005	53.182	12.525
0.01	1750	**0.996**	0.003	60.590	8.215
0.01	1900	**0.998**	0.002	78.510	22.452
0.01	2050	**0.998**	0.002	**79.863**	2.352

Table A.77: Detailed data for SBMClone - $\bar{p} = 0.010$.

Algorithm: SBMClone					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.015	100	-0.006	0.004	**0.075**	0.005
0.015	250	0.003	0.004	**0.411**	0.056
0.015	400	0.000	0.000	**1.173**	0.165
0.015	550	**0.822**	0.067	7.448	1.530
0.015	700	**0.948**	0.018	9.936	2.023
0.015	850	**0.976**	0.010	15.649	2.332
0.015	1000	**0.994**	0.006	23.356	4.858
0.015	1150	**0.992**	0.005	30.715	5.782
0.015	1300	**0.998**	0.002	40.576	4.562
0.015	1450	**1.000**	0.000	60.587	9.834
0.015	1600	**1.000**	0.000	69.065	2.422
0.015	1750	**1.000**	0.000	79.857	4.078
0.015	1900	**1.000**	0.000	123.345	13.337
0.015	2050	**1.000**	0.000	123.853	10.672

Table A.78: Detailed data for SBMClone - $\bar{p} = 0.015$.

Algorithm: SBMClone					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.02	100	0.001	0.017	0.089	0.006
0.02	250	0.000	0.001	**0.544**	0.069
0.02	400	**0.651**	0.365	3.640	1.326
0.02	550	**0.968**	0.009	11.351	5.417
0.02	700	**0.983**	0.006	11.450	0.653
0.02	850	**0.996**	0.004	22.585	4.789
0.02	1000	**0.999**	0.002	27.950	0.302
0.02	1150	**1.000**	0.000	46.888	5.476
0.02	1300	**1.000**	0.000	59.444	4.337
0.02	1450	**1.000**	0.000	93.819	55.277
0.02	1600	**1.000**	0.000	111.219	25.949
0.02	1750	**1.000**	0.000	112.792	14.307
0.02	1900	**1.000**	0.000	124.983	10.941
0.02	2050	**1.000**	0.000	174.558	56.477

Table A.79: Detailed data for SBMClone - $\bar{p} = 0.020$.

Algorithm: SBMClone					
$\bar{p}$	$m = n$	ARI		Runtime (s)	
		mean	SD	mean	SD
0.025	100	-0.000	0.010	0.091	0.007
0.025	250	0.000	0.001	0.865	0.333
0.025	400	**0.941**	0.019	5.447	2.104
0.025	550	**0.990**	0.004	9.720	2.928
0.025	700	**0.995**	0.005	15.559	1.329
0.025	850	**1.000**	0.000	32.544	10.743
0.025	1000	**1.000**	0.000	36.501	1.399
0.025	1150	**1.000**	0.000	48.175	2.975
0.025	1300	**1.000**	0.000	83.067	15.008
0.025	1450	**1.000**	0.000	117.497	30.489
0.025	1600	**1.000**	0.000	104.253	6.626
0.025	1750	**1.000**	0.000	118.230	8.490
0.025	1900	**1.000**	0.000	162.132	42.489
0.025	2050	**1.000**	0.000	274.658	126.202

Table A.80: Detailed data for SBMClone - $\bar{p} = 0.025$.

A.5 Mutation trees

A.5.1 Large number of cells

Algorithm: Naive					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
3	0.001	**0.053**	0.029	193.719	3.081
3	0.005	0.214	0.033	369.679	4.147
3	0.009	0.386	0.037	543.623	5.432
3	0.013	0.560	0.054	1087.781	521.344
3	0.017	0.621	0.011	1013.339	292.115
3	0.021	0.710	0.036	1045.185	3.075
3	0.025	0.773	0.017	1227.196	4.701
3	0.029	0.821	0.018	1852.683	670.465
3	0.033	0.869	0.025	1587.658	5.058
3	0.037	0.880	0.019	1746.534	5.077
3	0.041	0.907	0.015	1881.554	15.098

Table A.81: Detailed data for Naive - 3 clones.

Algorithm: Naive					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
4	0.001	**0.049**	0.012	191.752	0.854
4	0.005	0.153	0.023	352.969	2.523
4	0.009	0.318	0.032	511.869	2.843
4	0.013	0.396	0.039	671.182	3.367
4	0.017	0.430	0.024	822.315	4.553
4	0.021	0.491	0.050	981.317	5.946
4	0.025	0.547	0.026	1129.683	19.315
4	0.029	0.623	0.047	1284.842	11.574
4	0.033	0.637	0.049	**1437.998**	6.296
4	0.037	0.708	0.017	**1566.870**	43.876
4	0.041	0.724	0.048	1739.466	35.930

Table A.82: Detailed data for Naive - 4 clones.

Algorithm: Naive					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
6	0.001	**0.024**	0.002	185.746	1.340
6	0.005	0.105	0.016	309.651	1.845
6	0.009	0.209	0.014	**438.598**	4.706
6	0.013	0.279	0.022	561.541	1.383
6	0.017	0.330	0.014	684.371	2.655
6	0.021	0.395	0.020	806.675	7.102
6	0.025	0.449	0.041	919.354	17.647
6	0.029	0.494	0.014	1051.057	11.095
6	0.033	0.523	0.026	1172.686	4.357
6	0.037	0.547	0.015	**1290.358**	17.676
6	0.041	0.593	0.019	1415.166	10.761

Table A.83: Detailed data for Naive - 6 clones.

Algorithm: Naive					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
10	0.001	0.011	0.002	175.160	1.924
10	0.005	0.060	0.003	261.145	4.092
10	0.009	0.114	0.015	350.510	2.713
10	0.013	0.185	0.011	438.507	3.843
10	0.017	0.249	0.010	525.434	4.105
10	0.021	0.306	0.015	608.656	3.691
10	0.025	0.340	0.025	685.011	15.145
10	0.029	0.383	0.020	**774.698**	1.798
10	0.033	0.402	0.022	855.276	1.897
10	0.037	0.442	0.021	940.310	3.887
10	0.041	0.463	0.032	989.498	32.038

Table A.84: Detailed data for Naive - 10 clones.

Algorithm: Weighted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
3	0.001	0.007	0.002	404.137	22.814
3	0.005	0.619	0.118	422.242	4.058
3	0.009	0.888	0.046	613.994	10.963
3	0.013	0.957	0.009	1031.190	342.951
3	0.017	0.981	0.003	**1003.256**	9.114
3	0.021	0.979	0.001	1198.107	6.437
3	0.025	0.981	0.002	1422.921	6.803
3	0.029	0.980	0.001	1708.045	254.001
3	0.033	0.992	0.000	1828.617	7.143
3	0.037	0.993	0.000	2013.911	5.155
3	0.041	0.992	0.000	2178.509	14.510

Table A.85: Detailed data for Weighted - 3 clones.

Algorithm: Weighted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
4	0.001	0.010	0.002	662.838	74.151
4	0.005	0.248	0.230	404.585	1.375
4	0.009	0.687	0.052	581.735	1.776
4	0.013	0.844	0.008	755.522	2.348
4	0.017	0.886	0.029	927.125	1.756
4	0.021	0.919	0.017	1105.266	18.485
4	0.025	0.943	0.025	1308.142	3.918
4	0.029	0.964	0.016	1474.871	10.833
4	0.033	0.961	0.014	1653.590	4.339
4	0.037	0.965	0.013	1815.027	6.230
4	0.041	0.988	0.001	1994.106	44.021

Table A.86: Detailed data for Weighted - 4 clones.

Algorithm: Weighted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
6	0.001	0.007	0.001	1837.165	222.311
6	0.005	0.000	0.000	356.958	3.204
6	0.009	0.567	0.068	499.318	2.838
6	0.013	0.718	0.043	638.464	4.926
6	0.017	0.822	0.018	772.914	1.761
6	0.021	0.882	0.030	910.611	1.041
6	0.025	0.921	0.018	1048.991	6.414
6	0.029	0.944	0.011	1196.915	5.760
6	0.033	0.939	0.016	1338.422	1.461
6	0.037	0.963	0.018	1483.839	2.777
6	0.041	0.957	0.013	1614.548	5.548

Table A.87: Detailed data for Weighted - 6 clones.

Algorithm: Weighted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
10	0.001	0.008	0.002	9023.903	899.298
10	0.005	0.000	0.000	309.394	1.847
10	0.009	0.268	0.072	402.333	0.919
10	0.013	0.545	0.018	498.308	1.333
10	0.017	0.652	0.032	598.403	1.491
10	0.021	0.748	0.017	682.688	9.503
10	0.025	0.803	0.045	771.322	20.988
10	0.029	0.797	0.038	873.476	8.800
10	0.033	0.861	0.024	967.668	4.681
10	0.037	0.881	0.040	1063.175	4.028
10	0.041	0.917	0.020	1143.737	18.482

Table A.88: Detailed data for Weighted - 10 clones.

Algorithm: Adjusted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
3	0.001	0.007	0.003	455.430	23.224
3	0.005	0.801	0.152	431.546	1.609
3	0.009	0.982	0.003	649.911	3.771
3	0.013	0.996	0.001	862.698	10.317
3	0.017	0.999	0.001	1073.981	12.029
3	0.021	0.967	0.071	1320.306	79.309
3	0.025	0.969	0.065	1580.191	141.783
3	0.029	0.998	0.004	1845.086	120.231
3	0.033	0.998	0.002	2133.865	95.117
3	0.037	0.998	0.002	2368.153	158.443
3	0.041	0.980	0.038	2594.962	215.406

Table A.89: Detailed data for Adjusted - 3 clones.

Algorithm: Adjusted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
4	0.001	0.011	0.001	708.198	88.018
4	0.005	0.313	0.287	422.799	29.140
4	0.009	0.762	0.016	607.962	3.166
4	0.013	0.882	0.039	792.087	6.826
4	0.017	0.889	0.063	972.319	10.527
4	0.021	0.877	0.132	1178.722	33.203
4	0.025	0.900	0.075	1378.414	23.669
4	0.029	0.969	0.022	1551.623	12.035
4	0.033	0.971	0.016	1789.398	22.635
4	0.037	0.979	0.018	1971.302	40.132
4	0.041	0.982	0.019	2207.836	59.338

Table A.90: Detailed data for Adjusted - 4 clones.

Algorithm: Adjusted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
6	0.001	0.010	0.002	1927.244	228.235
6	0.005	0.000	0.000	417.224	5.426
6	0.009	0.668	0.050	507.224	2.710
6	0.013	0.762	0.071	657.922	16.764
6	0.017	0.802	0.111	799.810	25.628
6	0.021	0.850	0.099	937.476	35.839
6	0.025	0.854	0.117	1090.547	41.109
6	0.029	0.885	0.081	1268.501	83.273
6	0.033	0.893	0.080	1452.624	99.275
6	0.037	0.924	0.059	1659.692	163.888
6	0.041	0.936	0.042	1815.686	182.609

Table A.91: Detailed data for Adjusted - 6 clones.

Algorithm: Adjusted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
10	0.001	0.012	0.003	9291.932	977.650
10	0.005	0.000	0.000	372.267	2.905
10	0.009	0.317	0.090	406.582	4.258
10	0.013	0.610	0.031	507.245	4.533
10	0.017	0.687	0.024	603.224	5.216
10	0.021	0.755	0.025	686.980	17.491
10	0.025	0.788	0.036	775.545	19.560
10	0.029	0.832	0.041	878.237	16.580
10	0.033	0.881	0.056	972.478	9.350
10	0.037	0.884	0.022	1071.718	4.316
10	0.041	0.895	0.026	1153.180	35.891

Table A.92: Detailed data for Adjusted - 10 clones.

Algorithm: SBMClone					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
3	0.001	0.011	0.003	**13.196**	0.363
3	0.005	**0.999**	0.002	**194.646**	61.921
3	0.009	**1.000**	0.000	**288.268**	23.727
3	0.013	**1.000**	0.000	**454.445**	72.643
3	0.017	**1.000**	0.000	1590.455	2423.014
3	0.021	**1.000**	0.000	**689.551**	99.939
3	0.025	**1.000**	0.000	**880.533**	125.873
3	0.029	**1.000**	0.000	**1326.669**	822.728
3	0.033	**1.000**	0.000	**709.421**	12.256
3	0.037	**1.000**	0.000	**786.078**	35.150
3	0.041	**1.000**	0.000	**1029.837**	286.497

Table A.93: Detailed data for SBMClone - 3 clones.

Algorithm: SBMClone					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
4	0.001	0.016	0.003	**11.045**	0.814
4	0.005	**0.961**	0.002	**175.113**	72.701
4	0.009	**1.000**	0.001	**288.287**	55.608
4	0.013	**1.000**	0.000	**341.086**	12.252
4	0.017	**1.000**	0.000	**397.033**	12.590
4	0.021	**1.000**	0.000	**513.907**	93.354
4	0.025	**1.000**	0.000	**568.683**	32.000
4	0.029	**1.000**	0.000	**721.985**	44.668
4	0.033	**1.000**	0.000	1984.128	1886.507
4	0.037	**1.000**	0.000	1734.979	1177.768
4	0.041	**1.000**	0.000	**784.039**	15.171

Table A.94: Detailed data for SBMClone - 4 clones.

Algorithm: SBMClone					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
6	0.001	0.016	0.001	**7.150**	1.015
6	0.005	**0.788**	0.047	**135.298**	25.793
6	0.009	**0.995**	0.002	514.100	507.980
6	0.013	**1.000**	0.000	**260.016**	33.111
6	0.017	**1.000**	0.000	**357.723**	13.297
6	0.021	**1.000**	0.000	**451.265**	18.960
6	0.025	**1.000**	0.000	**511.703**	81.341
6	0.029	**1.000**	0.001	**582.309**	123.658
6	0.033	**1.000**	0.000	**689.552**	145.501
6	0.037	**1.000**	0.001	1525.383	1197.469
6	0.041	**1.000**	0.000	**897.345**	164.504

Table A.95: Detailed data for SBMClone - 6 clones.

Algorithm: SBMClone					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
10	0.001	**0.014**	0.002	**4.092**	0.438
10	0.005	**0.442**	0.064	**108.463**	34.610
10	0.009	**0.901**	0.010	**204.281**	135.795
10	0.013	**0.985**	0.004	**291.977**	70.228
10	0.017	**0.997**	0.002	**262.369**	57.374
10	0.021	**0.999**	0.001	**327.855**	56.091
10	0.025	**0.999**	0.001	**408.322**	105.318
10	0.029	**1.000**	0.000	1177.370	1731.318
10	0.033	**1.000**	0.000	**484.503**	142.189
10	0.037	**1.000**	0.001	**425.949**	112.582
10	0.041	**1.000**	0.001	**780.101**	642.350

Table A.96: Detailed data for SBMClone - 10 clones.

A.5.2 Medium number of cells

Algorithm: Naive					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
3	0.001	0.007	0.006	0.957	0.234
3	0.005	0.035	0.022	2.014	0.023
3	0.009	0.057	0.052	**2.759**	0.212
3	0.013	0.140	0.065	**3.404**	0.245
3	0.017	0.195	0.044	**4.424**	0.247
3	0.021	0.400	0.132	**5.170**	0.228
3	0.025	0.465	0.142	**6.251**	0.057
3	0.029	0.599	0.069	**7.081**	0.045
3	0.033	0.692	0.097	**7.718**	0.284
3	0.037	0.635	0.111	**8.691**	0.030
3	0.041	0.733	0.069	**9.289**	0.153

Table A.97: Detailed data for Naive - 3 clones.

Algorithm: Naive					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
4	0.001	0.009	0.007	1.856	0.224
4	0.005	**0.068**	0.028	3.314	0.249
4	0.009	0.110	0.015	**4.753**	0.241
4	0.013	0.154	0.030	**6.005**	0.218
4	0.017	0.277	0.043	**7.340**	0.211
4	0.021	0.291	0.047	**8.754**	0.234
4	0.025	0.321	0.071	**10.223**	0.257
4	0.029	0.400	0.055	**11.519**	0.265
4	0.033	0.448	0.072	**12.934**	0.290
4	0.037	0.587	0.024	**14.265**	0.320
4	0.041	0.556	0.077	**15.597**	0.272

Table A.98: Detailed data for Naive - 4 clones.

Algorithm: Naive					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
6	0.001	0.009	0.006	3.855	0.280
6	0.005	**0.044**	0.015	6.889	0.291
6	0.009	0.106	0.027	**9.347**	0.270
6	0.013	0.157	0.024	**11.818**	0.200
6	0.017	0.201	0.027	**14.163**	0.168
6	0.021	0.245	0.034	**16.721**	0.163
6	0.025	0.303	0.032	**19.106**	0.254
6	0.029	0.356	0.027	**21.653**	0.238
6	0.033	0.399	0.023	**24.058**	0.222
6	0.037	0.434	0.054	**26.651**	0.132
6	0.041	0.469	0.031	**28.927**	0.115

Table A.99: Detailed data for Naive - 6 clones.

Algorithm: Naive					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
10	0.001	0.005	0.002	10.943	0.244
10	0.005	**0.040**	0.006	16.113	0.248
10	0.009	0.068	0.011	**21.245**	0.319
10	0.013	0.103	0.016	**26.058**	0.353
10	0.017	0.160	0.019	**31.409**	0.435
10	0.021	0.191	0.020	**36.359**	0.446
10	0.025	0.243	0.023	**40.668**	0.405
10	0.029	0.306	0.015	**45.661**	0.543
10	0.033	0.320	0.026	**50.257**	0.549
10	0.037	0.363	0.017	**55.060**	0.586
10	0.041	0.394	0.025	**59.795**	0.631

Table A.100: Detailed data for Naive - 10 clones.

Algorithm: Weighted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
3	0.001	0.001	0.004	21.606	2.627
3	0.005	**0.051**	0.034	**1.932**	0.257
3	0.009	0.136	0.053	2.999	0.046
3	0.013	0.283	0.131	3.844	0.026
3	0.017	0.404	0.182	4.644	0.221
3	0.021	0.625	0.037	5.530	0.061
3	0.025	0.690	0.119	6.484	0.063
3	0.029	0.872	0.042	7.179	0.259
3	0.033	0.907	0.021	8.216	0.103
3	0.037	0.884	0.067	8.886	0.241
3	0.041	0.905	0.028	9.618	0.245

Table A.101: Detailed data for Weighted - 3 clones.

Algorithm: Weighted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
4	0.001	0.002	0.006	50.498	5.740
4	0.005	0.062	0.018	3.425	0.222
4	0.009	0.144	0.024	4.933	0.269
4	0.013	0.323	0.043	6.409	0.262
4	0.017	0.446	0.151	7.842	0.278
4	0.021	0.523	0.065	9.316	0.303
4	0.025	0.653	0.112	10.781	0.241
4	0.029	0.779	0.050	12.041	0.311
4	0.033	0.847	0.072	13.533	0.264
4	0.037	0.873	0.033	14.797	0.245
4	0.041	0.901	0.013	16.178	0.352

Table A.102: Detailed data for Weighted - 4 clones.

Algorithm: Weighted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
6	0.001	0.003	0.004	153.375	18.201
6	0.005	0.042	0.026	7.109	0.053
6	0.009	0.185	0.013	9.958	0.262
6	0.013	0.282	0.030	12.520	0.325
6	0.017	0.400	0.026	15.175	0.277
6	0.021	0.460	0.050	17.660	0.245
6	0.025	0.516	0.051	20.235	0.437
6	0.029	0.652	0.087	22.829	0.310
6	0.033	0.766	0.073	25.722	0.401
6	0.037	0.814	0.069	28.219	0.188
6	0.041	0.851	0.090	30.928	0.299

Table A.103: Detailed data for Weighted - 6 clones.

Algorithm: Weighted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
10	0.001	0.004	0.003	828.935	80.352
10	0.005	0.006	0.014	17.821	0.362
10	0.009	0.178	0.027	23.068	0.405
10	0.013	0.286	0.021	28.459	0.294
10	0.017	0.368	0.021	34.168	0.256
10	0.021	0.434	0.031	39.033	0.335
10	0.025	0.503	0.024	44.547	0.294
10	0.029	0.590	0.032	49.472	0.354
10	0.033	0.650	0.026	54.525	0.328
10	0.037	0.712	0.055	60.172	0.170
10	0.041	0.745	0.021	64.803	0.450

Table A.104: Detailed data for Weighted - 10 clones.

Algorithm: Adjusted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
3	0.001	0.010	0.015	22.769	2.610
3	0.005	0.018	0.028	2.255	0.030
3	0.009	**0.214**	0.045	3.127	0.240
3	0.013	0.519	0.108	4.307	0.119
3	0.017	0.808	0.090	5.303	0.101
3	0.021	0.953	0.022	6.313	0.231
3	0.025	0.881	0.146	7.372	0.668
3	0.029	0.982	0.021	8.750	0.443
3	0.033	0.986	0.011	9.947	0.524
3	0.037	0.974	0.005	11.522	1.605
3	0.041	0.976	0.006	13.669	3.385

Table A.105: Detailed data for Adjusted - 3 clones.

Algorithm: Adjusted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
4	0.001	0.011	0.010	52.718	5.742
4	0.005	0.034	0.022	3.603	0.243
4	0.009	**0.297**	0.075	5.351	0.277
4	0.013	0.531	0.054	6.827	0.199
4	0.017	0.728	0.072	8.402	0.318
4	0.021	0.822	0.027	10.033	0.328
4	0.025	0.886	0.023	11.674	0.202
4	0.029	0.944	0.010	13.799	0.710
4	0.033	0.942	0.020	15.497	0.578
4	0.037	0.966	0.019	17.200	1.243
4	0.041	0.957	0.026	19.008	1.708

Table A.106: Detailed data for Adjusted - 4 clones.

Algorithm: Adjusted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
6	0.001	0.013	0.008	159.584	18.375
6	0.005	0.023	0.020	7.499	0.324
6	0.009	0.272	0.048	10.389	0.309
6	0.013	0.474	0.032	13.161	0.196
6	0.017	0.639	0.066	15.909	0.330
6	0.021	0.762	0.014	18.869	0.319
6	0.025	0.840	0.073	21.641	0.376
6	0.029	0.907	0.035	24.517	0.260
6	0.033	0.941	0.007	28.004	0.596
6	0.037	0.965	0.008	31.137	0.317
6	0.041	0.964	0.006	34.729	0.627

Table A.107: Detailed data for Adjusted - 6 clones.

Algorithm: Adjusted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
10	0.001	0.009	0.005	864.141	82.437
10	0.005	0.005	0.012	20.101	1.072
10	0.009	0.271	0.015	23.795	0.295
10	0.013	0.411	0.038	29.523	0.459
10	0.017	0.543	0.012	35.206	0.334
10	0.021	0.621	0.035	40.397	0.447
10	0.025	0.702	0.016	45.951	0.348
10	0.029	0.774	0.027	51.645	0.610
10	0.033	0.803	0.034	57.063	0.785
10	0.037	0.821	0.034	62.711	0.895
10	0.041	0.921	0.022	68.465	1.335

Table A.108: Detailed data for Adjusted - 10 clones.

Algorithm: SBMClone					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
3	0.001	**0.012**	0.008	**0.369**	0.070
3	0.005	0.000	0.000	2.110	0.268
3	0.009	0.000	0.000	7.235	2.536
3	0.013	**0.966**	0.022	22.328	4.758
3	0.017	**1.000**	0.000	31.939	11.554
3	0.021	**1.000**	0.000	26.761	5.708
3	0.025	**1.000**	0.000	34.520	9.937
3	0.029	**1.000**	0.000	37.931	6.075
3	0.033	**1.000**	0.000	48.189	17.625
3	0.037	**1.000**	0.000	49.956	6.693
3	0.041	**1.000**	0.000	56.873	5.414

Table A.109: Detailed data for SBMClone - 3 clones.

Algorithm: SBMClone					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
4	0.001	**0.018**	0.007	**0.444**	0.073
4	0.005	0.000	0.000	**2.994**	0.380
4	0.009	0.214	0.293	11.563	5.353
4	0.013	**0.762**	0.091	24.441	7.011
4	0.017	**0.845**	0.078	36.847	6.819
4	0.021	**0.978**	0.022	37.720	12.017
4	0.025	**0.999**	0.003	62.697	28.498
4	0.029	**1.000**	0.000	55.856	14.989
4	0.033	**1.000**	0.000	62.183	19.687
4	0.037	**1.000**	0.000	79.421	21.563
4	0.041	**1.000**	0.000	69.033	8.428

Table A.110: Detailed data for SBMClone - 4 clones.

Algorithm: SBMClone					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
6	0.001	**0.016**	0.004	**0.653**	0.179
6	0.005	0.000	0.000	**3.940**	0.651
6	0.009	**0.451**	0.009	29.086	14.691
6	0.013	**0.653**	0.117	44.387	17.202
6	0.017	**0.862**	0.061	42.123	10.741
6	0.021	**0.912**	0.051	65.206	32.534
6	0.025	**0.986**	0.021	51.962	14.981
6	0.029	**0.999**	0.002	57.030	16.539
6	0.033	**1.000**	0.000	80.302	22.808
6	0.037	**1.000**	0.000	91.472	31.036
6	0.041	**0.999**	0.002	83.011	17.699

Table A.111: Detailed data for SBMClone - 6 clones.

Algorithm: SBMClone					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
10	0.001	**0.013**	0.002	**0.748**	0.070
10	0.005	0.000	0.000	**6.992**	0.520
10	0.009	**0.341**	0.003	40.943	17.789
10	0.013	**0.585**	0.007	33.262	5.900
10	0.017	**0.788**	0.007	53.347	19.141
10	0.021	**0.804**	0.003	46.598	8.908
10	0.025	**0.832**	0.036	79.401	24.228
10	0.029	**0.846**	0.044	87.434	27.430
10	0.033	**0.990**	0.015	107.302	36.723
10	0.037	**0.999**	0.001	91.504	23.133
10	0.041	**1.000**	0.000	85.345	4.243

Table A.112: Detailed data for SBMClone - 10 clones.

A.5.3 Small number of cells

Algorithm: Naive					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
3	0.001	**0.028**	0.039	**0.129**	0.002
3	0.005	0.039	0.024	**0.471**	0.009
3	0.009	0.271	0.094	**0.829**	0.011
3	0.013	0.286	0.141	**1.204**	0.009
3	0.017	0.414	0.146	**1.534**	0.021
3	0.021	0.347	0.036	**1.951**	0.018
3	0.025	0.431	0.104	**2.344**	0.041
3	0.029	0.479	0.124	**2.748**	0.036
3	0.033	0.421	0.109	**3.089**	0.123
3	0.037	0.372	0.000	**3.519**	0.053
3	0.041	0.372	0.000	**4.027**	0.044

Table A.113: Detailed data for Naive - 3 clones.

Algorithm: Naive					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
4	0.001	0.032	0.028	**0.194**	0.006
4	0.005	0.141	0.076	**0.740**	0.007
4	0.009	0.281	0.145	**1.319**	0.015
4	0.013	0.354	0.083	**1.926**	0.016
4	0.017	0.425	0.027	**2.522**	0.017
4	0.021	0.435	0.037	**3.107**	0.038
4	0.025	0.498	0.006	**3.739**	0.036
4	0.029	0.500	0.005	**4.332**	0.027
4	0.033	0.496	0.001	**5.041**	0.047
4	0.037	0.496	0.001	**5.721**	0.056
4	0.041	0.495	0.000	**6.364**	0.079

Table A.114: Detailed data for Naive - 4 clones.

Algorithm: Naive					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
6	0.001	**0.020**	0.015	**0.384**	0.091
6	0.005	0.083	0.035	**1.349**	0.087
6	0.009	0.212	0.041	**2.308**	0.080
6	0.013	0.275	0.032	**3.368**	0.036
6	0.017	0.367	0.062	**4.464**	0.040
6	0.021	0.480	0.055	**5.539**	0.080
6	0.025	0.592	0.048	**6.628**	0.074
6	0.029	0.563	0.054	**7.723**	0.099
6	0.033	0.568	0.040	**8.788**	0.164
6	0.037	0.612	0.012	**9.953**	0.156
6	0.041	0.615	0.013	**10.665**	0.473

Table A.115: Detailed data for Naive - 6 clones.

Algorithm: Naive					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
10	0.001	**0.019**	0.007	0.931	0.020
10	0.005	0.059	0.011	**2.750**	0.024
10	0.009	0.140	0.016	**4.614**	0.078
10	0.013	0.248	0.017	**6.445**	0.085
10	0.017	0.300	0.028	**8.505**	0.060
10	0.021	0.366	0.038	**10.801**	0.153
10	0.025	0.419	0.033	**12.866**	0.100
10	0.029	0.475	0.055	**14.791**	0.068
10	0.033	0.555	0.020	**16.995**	0.197
10	0.037	0.536	0.031	**18.824**	0.139
10	0.041	0.565	0.009	**19.965**	0.751

Table A.116: Detailed data for Naive - 10 clones.

Algorithm: Weighted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
3	0.001	0.002	0.021	0.275	0.068
3	0.005	0.093	0.057	0.492	0.007
3	0.009	0.209	0.053	0.874	0.012
3	0.013	0.401	0.147	1.301	0.011
3	0.017	0.388	0.028	1.681	0.023
3	0.021	0.474	0.129	2.193	0.029
3	0.025	0.377	0.011	2.634	0.051
3	0.029	0.372	0.000	3.121	0.056
3	0.033	0.421	0.109	3.493	0.019
3	0.037	0.372	0.000	4.099	0.073
3	0.041	0.372	0.000	4.764	0.044

Table A.117: Detailed data for Weighted - 3 clones.

Algorithm: Weighted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
4	0.001	**0.033**	0.033	0.538	0.173
4	0.005	0.111	0.051	0.781	0.008
4	0.009	0.320	0.124	1.468	0.116
4	0.013	0.390	0.079	2.095	0.026
4	0.017	0.476	0.031	2.865	0.138
4	0.021	0.500	0.013	3.495	0.061
4	0.025	0.494	0.013	4.329	0.129
4	0.029	0.495	0.000	4.991	0.040
4	0.033	0.495	0.000	5.961	0.097
4	0.037	0.495	0.000	6.766	0.131
4	0.041	0.495	0.000	7.524	0.143

Table A.118: Detailed data for Weighted - 4 clones.

Algorithm: Weighted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
6	0.001	0.004	0.006	1.073	0.109
6	0.005	0.092	0.012	1.386	0.018
6	0.009	0.258	0.048	2.496	0.057
6	0.013	0.345	0.052	3.773	0.103
6	0.017	0.488	0.059	4.946	0.072
6	0.021	0.559	0.033	6.258	0.103
6	0.025	0.601	0.016	7.576	0.042
6	0.029	0.612	0.013	8.908	0.104
6	0.033	0.617	0.007	10.251	0.166
6	0.037	0.615	0.010	11.598	0.100
6	0.041	0.623	0.000	12.543	0.688

Table A.119: Detailed data for Weighted - 6 clones.

Algorithm: Weighted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
10	0.001	0.011	0.007	2.368	0.201
10	0.005	0.081	0.042	2.789	0.035
10	0.009	0.215	0.033	5.058	0.092
10	0.013	0.350	0.022	7.446	0.154
10	0.017	0.425	0.034	9.625	0.064
10	0.021	0.485	0.022	12.226	0.056
10	0.025	0.560	0.028	14.880	0.169
10	0.029	0.598	0.020	17.104	0.207
10	0.033	0.610	0.008	19.676	0.192
10	0.037	0.612	0.022	22.070	0.342
10	0.041	0.621	0.003	23.335	1.095

Table A.120: Detailed data for Weighted - 10 clones.

Algorithm: Adjusted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
3	0.001	0.008	0.024	0.316	0.070
3	0.005	**0.138**	0.072	0.612	0.016
3	0.009	**0.478**	0.153	1.137	0.114
3	0.013	**0.873**	0.064	1.837	0.282
3	0.017	**0.951**	0.069	2.433	0.328
3	0.021	**0.883**	0.235	3.346	0.918
3	0.025	**0.970**	0.028	3.885	0.635
3	0.029	**0.920**	0.044	4.833	0.599
3	0.033	**0.919**	0.045	5.663	0.663
3	0.037	**0.911**	0.041	6.997	0.590
3	0.041	**0.864**	0.062	8.305	0.367

Table A.121: Detailed data for Adjusted - 3 clones.

Algorithm: Adjusted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
4	0.001	0.019	0.014	0.535	0.098
4	0.005	**0.174**	0.081	0.905	0.016
4	0.009	**0.359**	0.069	1.736	0.153
4	0.013	**0.582**	0.060	2.598	0.183
4	0.017	**0.776**	0.102	3.656	0.491
4	0.021	**0.833**	0.137	4.861	1.010
4	0.025	**0.904**	0.058	5.996	0.820
4	0.029	**0.831**	0.145	7.801	1.552
4	0.033	**0.775**	0.132	9.368	1.630
4	0.037	**0.762**	0.165	11.301	2.064
4	0.041	**0.824**	0.110	12.461	1.483

Table A.122: Detailed data for Adjusted - 4 clones.

Algorithm: Adjusted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
6	0.001	0.005	0.005	1.122	0.107
6	0.005	**0.124**	0.040	1.590	0.040
6	0.009	**0.297**	0.034	2.827	0.063
6	0.013	**0.478**	0.078	4.320	0.132
6	0.017	**0.651**	0.091	6.019	0.345
6	0.021	**0.785**	0.052	8.431	1.339
6	0.025	**0.793**	0.122	10.732	1.642
6	0.029	0.768	0.138	14.243	2.747
6	0.033	**0.710**	0.101	17.219	1.968
6	0.037	**0.694**	0.105	20.293	2.405
6	0.041	**0.653**	0.108	22.407	3.434

Table A.123: Detailed data for Adjusted - 6 clones.

Algorithm: Adjusted					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
10	0.001	0.011	0.007	2.503	0.191
10	0.005	**0.094**	0.016	3.063	0.136
10	0.009	**0.293**	0.028	5.601	0.264
10	0.013	**0.436**	0.046	8.199	0.186
10	0.017	**0.591**	0.038	11.574	0.555
10	0.021	**0.676**	0.045	15.222	1.303
10	0.025	**0.785**	0.066	19.659	2.273
10	0.029	**0.763**	0.040	24.049	2.239
10	0.033	**0.809**	0.072	28.623	2.729
10	0.037	0.791	0.048	35.034	2.852
10	0.041	0.755	0.085	38.535	4.095

Table A.124: Detailed data for Adjusted - 10 clones.

Algorithm: SBMClone					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
3	0.001	0.000	0.000	0.522	0.092
3	0.005	0.000	0.000	2.482	0.231
3	0.009	0.000	0.000	6.213	0.274
3	0.013	0.000	0.000	188.776	223.519
3	0.017	0.000	0.000	134.597	119.107
3	0.021	0.225	0.308	50.808	20.751
3	0.025	0.563	0.000	105.665	32.605
3	0.029	0.225	0.308	64.030	25.255
3	0.033	0.113	0.252	51.972	55.580
3	0.037	0.113	0.252	46.033	35.461
3	0.041	0.000	0.000	31.533	22.493

Table A.125: Detailed data for SBMClone - 3 clones.

Algorithm: SBMClone					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
4	0.001	0.000	0.000	0.537	0.098
4	0.005	0.000	0.000	3.671	0.448
4	0.009	0.000	0.000	9.301	0.292
4	0.013	0.000	0.000	61.037	42.378
4	0.017	0.367	0.251	74.405	37.769
4	0.021	0.466	0.301	113.247	53.388
4	0.025	0.206	0.313	81.120	37.027
4	0.029	0.335	0.473	65.845	28.904
4	0.033	0.460	0.472	117.456	89.308
4	0.037	0.029	0.064	35.053	12.284
4	0.041	0.065	0.145	41.295	13.135

Table A.126: Detailed data for SBMClone - 4 clones.

Algorithm: SBMClone					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
6	0.001	0.000	0.000	0.781	0.098
6	0.005	0.000	0.000	6.410	0.610
6	0.009	0.000	0.000	258.016	218.065
6	0.013	0.234	0.134	101.359	84.888
6	0.017	0.386	0.125	113.737	26.962
6	0.021	0.511	0.184	168.459	38.403
6	0.025	0.610	0.099	189.863	41.964
6	0.029	**0.799**	0.041	219.038	95.052
6	0.033	0.477	0.267	190.463	46.873
6	0.037	0.446	0.171	127.324	20.030
6	0.041	0.332	0.071	241.127	207.067

Table A.127: Detailed data for SBMClone - 6 clones.

Algorithm: SBMClone					
clones	$\bar{p}$	ARI		Runtime (s)	
		mean	SD	mean	SD
10	0.001	0.000	0.000	**0.907**	0.075
10	0.005	0.000	0.000	9.489	0.623
10	0.009	0.079	0.050	3207.806	4004.426
10	0.013	0.404	0.181	114.453	23.322
10	0.017	0.442	0.100	111.900	13.544
10	0.021	0.429	0.129	172.250	37.933
10	0.025	0.470	0.113	219.143	64.845
10	0.029	0.436	0.147	287.749	21.800
10	0.033	0.693	0.158	284.698	63.323
10	0.037	**0.807**	0.083	396.343	59.608
10	0.041	**0.942**	0.057	460.582	93.404

Table A.128: Detailed data for SBMClone - 10 clones.

A.6 Sampling efficiency

Algorithm: Adjusted				
samples	ARI		Runtime (s)	
	mean	SD	mean	SD
0	0.847	0.064	301.499	21.206
1	0.967	0.006	298.270	9.511
2	0.965	0.012	314.851	20.542
4	0.956	0.012	315.613	29.114
10	0.964	0.003	324.534	34.773
21	0.959	0.006	335.625	17.620
46	0.956	0.015	386.866	9.888
100	0.967	0.010	477.156	14.030
215	0.961	0.012	697.197	90.912
464	0.965	0.006	1116.324	75.218
1000	0.964	0.006	2132.039	97.754

Table A.129: Detailed data for sampling efficiency tests. $m = 4000$, $n = 5000$, $\bar{p} = 0.005$, 30% overlap. Using zero samples is equivalent to using the weighted algorithm.

A.7 Realistic parameters

Algorithm: Naive				
$\bar{p}$	ARI		Runtime (s)	
	mean	SD	mean	SD
0.001	-0.017	0.005	**0.089**	0.001
0.005	0.108	0.170	**0.302**	0.004
0.009	0.369	0.211	**0.512**	0.013
0.013	0.477	0.053	**0.756**	0.006
0.017	0.515	0.029	**0.939**	0.007
0.021	0.515	0.029	**1.212**	0.005
0.025	0.528	0.000	**1.414**	0.016
0.029	0.528	0.000	**1.648**	0.013
0.033	0.528	0.000	**1.852**	0.011
0.037	0.528	0.000	**2.082**	0.026
0.041	0.528	0.000	**2.482**	0.015

Table A.130: Detailed data for Naive - realistic parameters.

Algorithm: Weighted				
$\bar{p}$	ARI		Runtime (s)	
	mean	SD	mean	SD
0.001	-0.015	0.006	0.211	0.041
0.005	0.021	0.053	0.315	0.003
0.009	0.370	0.216	0.531	0.013
0.013	0.476	0.029	0.803	0.011
0.017	0.515	0.029	1.003	0.008
0.021	0.515	0.029	1.297	0.015
0.025	0.515	0.029	1.544	0.023
0.029	0.528	0.000	1.838	0.014
0.033	0.528	0.000	2.105	0.031
0.037	0.528	0.000	2.402	0.037
0.041	0.528	0.000	2.910	0.034

Table A.131: Detailed data for Weighted - realistic parameters.

Algorithm: Adjusted				
$\bar{p}$	ARI		Runtime (s)	
	mean	SD	mean	SD
0.001	**0.026**	0.060	0.245	0.035
0.005	**0.354**	0.123	0.436	0.054
0.009	**0.931**	0.111	0.703	0.045
0.013	**0.982**	0.040	1.180	0.157
0.017	**1.000**	0.000	1.658	0.292
0.021	**1.000**	0.000	2.034	0.258
0.025	**1.000**	0.000	2.521	0.347
0.029	**1.000**	0.000	3.293	0.542
0.033	**1.000**	0.000	4.367	0.407
0.037	**0.982**	0.040	5.134	0.375
0.041	**0.964**	0.049	6.318	0.120

Table A.132: Detailed data for Adjusted - realistic parameters.

Algorithm: SBMClone				
$\bar{p}$	ARI		Runtime (s)	
	mean	SD	mean	SD
0.001	0.000	0.000	0.469	0.219
0.005	0.000	0.000	2.244	1.982
0.009	0.000	0.000	4.458	0.178
0.013	0.000	0.000	104.828	52.404
0.017	0.000	0.000	60.222	54.549
0.021	0.000	0.000	464.168	879.135
0.025	0.000	0.000	155.834	151.750
0.029	0.000	0.000	275.595	433.404
0.033	0.000	0.000	41.353	2.184
0.037	0.000	0.000	27.433	15.362
0.041	0.200	0.447	26.943	14.296

Table A.133: Detailed data for SBMClone - realistic parameters.