UNIVERSITÀ DEGLI STUDI DI PADOVA

Department of Information Engineering

Master Course in Computer Engineering

Computational methods to characterize mutations in cancer phylogenetic trees

Graduate :
Tommaso SCARPA

Supervisor:
Fabio VANDIN

Academic Year 2016-2017

Abstract

This thesis introduces the problem of reconstructing the evolutionary tree from single-cell sequencing data and the following problem of characterization of DNA mutations.

The objective is to understand if it is possible to characterize such mutations relying only on the reconstructed phylogenetic tree structure. This is done by first implementing a software that simulates the growth of a potential tumoral mass starting from a single cell, in order to be able to simulate a real size phylogenetic tree. On this tree are then executed the Two Proportion Z Test, the Two Sample Kolmogorov-Smirnov Test and the 2 Sample Anderson-Darling Test. These tests are experimented on instances with different specifics in order to find best and worst application scenarios for each one of them.

The final results are encouraging and show that characterizing mutations relying solely on a tree structure can be possible and there should be high margins of improvement.

Contents

1 Introduction

1.1 Background

Our bodies are amazing proofs of the power of teamwork. Our cells work together in symbiosis in order to let us live our frenetic 21st century lives. What we consider so common that we barely pay attention (like breathing or our heart beating), is in fact the the result of millions of cells fulfilling their purpose and working together.

Every cell has a sort of instruction set, called DNA. The DNA is a double helix shaped molecule that carries the genetic instructions used in the growth, development, functioning and reproduction of all known living organisms and many viruses. The two DNA strands are composed of units called nucleotides. There are four type of nucleotides (named by the kind of nitrogenous base they incorporate): cytosine (C), guanine (G), adenine (A) and thymine (T). These bases go in pairs along the two helices, in such a way that if a certain place contains a C nucleotide, the corresponding place on the other helix will contain an A and vice versa. The same goes for G and T nucleotides.

The process of reading a substring of DNA can be, unfortunately, prone to errors. Most of the time these errors do not have any repercussions, but sometimes they

can bring to very dangerous situations. One very delicate situation is the moment when a cell reproduces. During this process a single cell has to duplicate each one of its subcellular components and all its genetic code in order to create a copy of itself.
During the duplication process of the DNA string there is the chance that some nucleotides are not copied correctly generating some anomalies in the genetic code (mutations). In most cases this won't have any negative effects. The mutations might appear in segments of DNA that are not read during the cell life or, even if that's the case, they could be harmless due to some redundancy in the string that could help reducing the effects of the mutation or again they could trigger some new behavior so abnormal to cause the cell to die or be killed by the organism's immune system. There are some cases though in which these mutations, especially when accumulated, can give the cell some advantages; advantages that could let it escape or even resist the immune system response, causing it to reproduce and live more easily than its pairs. It is in these cases, that we talk about tumoral or cancerous cells.

1.2 Tumoral evolution and Motivation

Tumor development can be described as a dynamic evolutionary process acting at the level of individual cells, in which a cell population accumulates mutations over time and evolves into a mix of genetically distinct subpopulations, called clones [1]. A tumor usually derives from a single ancestor cell whose unique set of genetic material conferred it a growth advantage over its pairs and helped it evade the body's immune system response. Consequently the clones derived from this cells acquire the same advantages and will develop furhter into more subclones, accumulating additional mutations. In this tumoral environment different clones compete against each other for resources, while the most successful

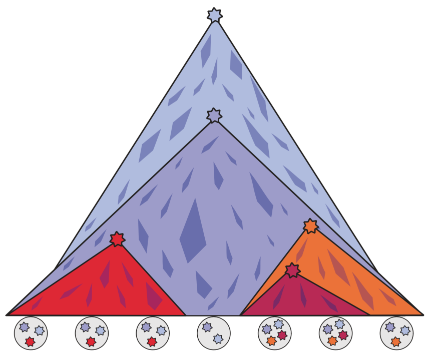


Figure 1.1. *Example of tumor evolution (taken from [3]).*

ones will replace the others, until they themselves are replaced by other clones [2] (Figure 1.1).

The genetic diversity that arises from this process inside the tumoral environment is believed to be one of the main causes of treatment failure and relapse. This is because usually the drug targets the dominant clone at the moment of the diagnosis and, once it has been eliminated, there might be an expansion of other subclones not targeted by the drug or an arising of drug resistant subclones.

Understanding the evolution of the tumoral growth can help in creating new drugs able to target the most effective clones or perform preemptive treatment in patients with similar histories [4] [5] [6].

Since every tumoral cell is related to the others via a binary genealogical tree it has become more and more important nowadays to find a method to reconstruct such tree from the data obtained from the sequencing experiments. Modern DNA sequencing technologies have improved to the point that is now possible to sequence the DNA of a tumoral mass with the precision of a single cell. The so called single-cell sequencing technologies, however, have the major drawback of being still very prone to errors compared to other older sequencing techniques. Usually the false positives rate varies from 2.67×10^{-5} to 6.7×10^{-5} [7] [8] [9], meaning that the number of false positives could outnumber true somatic variants

[10], while the false negatives rate, mostly due to allele dropout, varies from 0.16 to 0.43 [7] [8] [9]. Related to this there is the issue of missing values, that occurs when all copies of a genetic locus fail to amplify, unfortunately a very common problem in this kind of datasets [7] [8] [9].
Another
challenge when facing this kind of data lies in unobserved subpopulations. Due to sampling biases, undersampling or subpopulations extinction it is very likely for the sampled cells to be only a fraction of the subpopulations that evolved in the tumoral environment. This has to be taken into account when trying to reconstruct the phylogenetic tree.
Lastly one of the biggest issues when using this datasets is their number and size. Of the few datasets that are available, most of them contain data from not more than 60 cells [7] [8] [9]. Although the problem of tree reconstruction is not trivial even with datasets this small, it is easy to see that they do not reflect the size of a real tumoral mass. So the statistical value of the results on those datasets is yet to be proved.

Despite the issues described so far, more and more efforts are being put on the topic of reconstructing the phylogenetic tree of a tumoral mass from single-cell sequencing data. However, at the moment of the writing of this thesis, no one is addressing an issue that is strictly linked to the reconstruction of phylogenetic trees. After having inferred the tree that better describes the history of the tumoral mass, there is no information about which of the mutations that occurred during the history of such tumor are the ones responsible for the disease. In other words there is no mean to differentiate between *passenger* (harmless) and *driver* (potentially carcinogenic) mutations once the tree is reconstructed.

Seeing this gap we thought to address the issue and try to understand how one could characterize such mutations based on the study of the phylogenetic tree. One of the purposes of this thesis is, in fact, to understand if it is possible to differentiate between harmless and potentially harmful mutations by looking at the behavior of the phylogenetic tree.

2 State of the Art

In this chapter we'll describe the latest algorithms used to rebuild such a tree from single-cell sequencing data of a tumoral mass. In particular we'll confront two of the most recent algorithms developed on such topic: SCITE [3] and OncoNEM [11]. For each algorithm we'll describe the model they use followed by a brief report on their conclusions. We'll then conclude with a brief comparison of the two algorithms.

2.1 Single Cell Inference of Tumor Evolution - SCITE

SCITE is a stochastic search algorithm capable of identifying the evolutionary history of a tumor even from noisy and incomplete mutation profiles of single cells. The algorithm uses a flexible Markov Chain Monte Carlo (MCMC) sampling scheme that allows: to compute the maximum-likelihood mutation history, to sample from the posterior probability distribution, and to estimate the error rates of the underlying sequencing experiments.
The only required assumptions are the restriction of the evolutionary model to point mutations (single nucleotide base mutations in the form of insertion/deletion or substitution) and the infinite sites assumption (which states that every genome

position mutates at most once in the evolutionary history of a tumor).

2.1.1 Model of tumor evolution and tree representation

To describe the mutation status of m single cells at n different loci they use a binary $n \times m$ mutation matrix E. In this matrix a 1, respectively a 0, at entry (i, j) denotes the presence, respectively the absence, of mutation i in cell j (Figure 2.1b). With the exclusion of convergent evolution (that is the process by which two unrelated species evolve similar characteristics), due to the infinite sites assumption, this matrix defines a perfect phylogeny of the single cells. This means that there exists a rooted binary tree, with the cells as leaves, in which every mutation can be placed on one edge, such that the mutation status of every leaf equals the set of mutations on its path to the root (Figure 2.1a). To simplify the notation mutations present in all cells and mutations observed only in a single cell can be removed from the data as their location in the tree does not bring any relevant information. So example, the mutation matrix shown in Figure 2.1b can be reduced to:

$$\mathrm{E} = \begin{array}{c} \\ M_1 \\ M_2 \\ M_3 \end{array} \begin{array}{c} \begin{array}{ccccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_7 & s_6 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array} \tag{2.1}$$

where each line represents a different mutation M_1, M_2 and M_3 and each column represents one cell. In general, the binary tree defined by the matrix E will not be unique. In the tree in Figure 2.1a, since the three leftmost leaves all have the same mutation status, they can be interchanged arbitrarily. We can also represent E more compactly as a mutation tree T (Figure 2.1c), which represents the mutations as nodes and connects them according to their order in the evolutionary

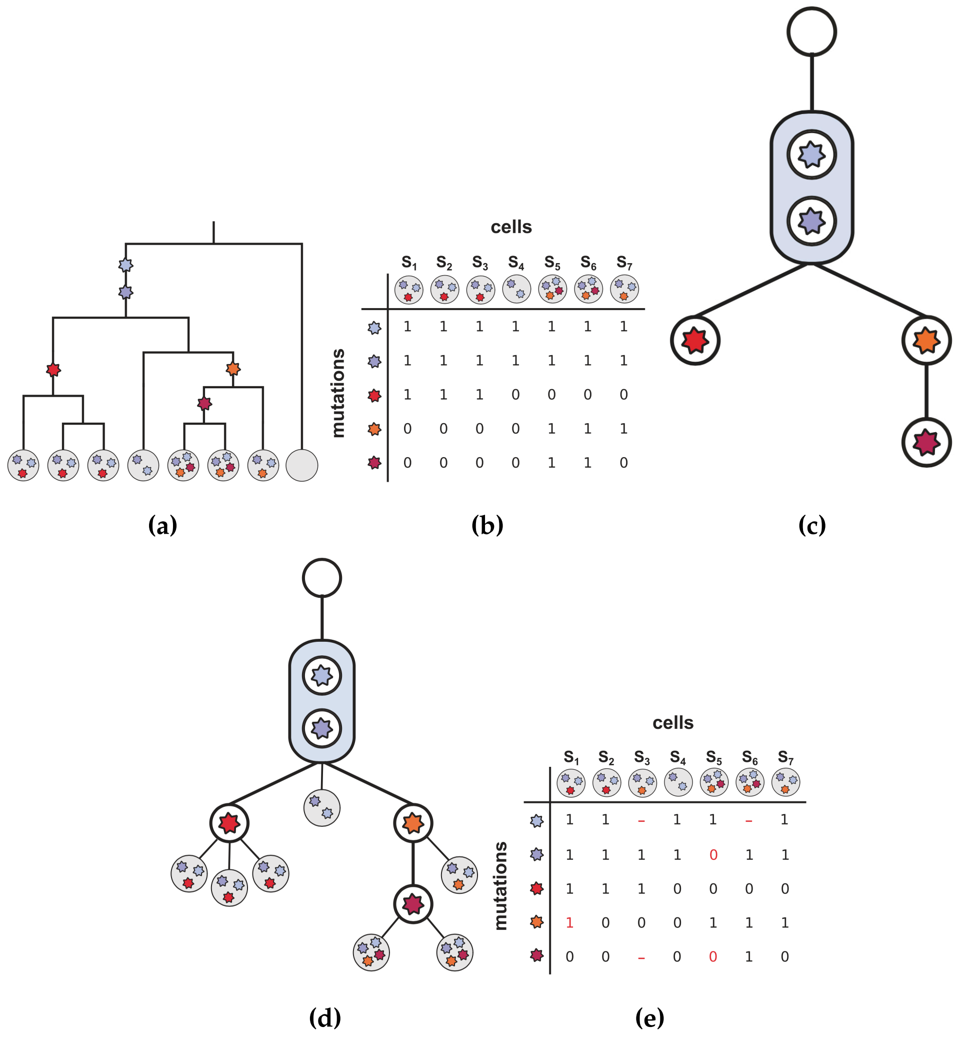


Figure 2.1. *(**a**) Binary genealogical tree of sequenced cells. (**b**) Binary mutation matrix representing the mutation status of sequenced tumor cells. (**c**) Representation of a mutation tree. (**d**) Representation of a mutation tree with single-cell samples attached. (**e**) Representation of a noisy mutation matrix with missing values. Figures taken from [3].*

history. The empty node indicates the root. The mutation tree is in fact very similar to the perfect phylogeny tree, where instead of placing the mutations along the edges they are encapsulated inside the nodes. This mutation tree can be augmented with the sequenced cells by attaching them to the node that matches their mutation state (Figure 2.1d). In this kind of trees the order of mutations that are shared by the exact same set of cells is unidentifiable (see the two mutations grouped in the bigger node).

A rooted mutation tree T over n mutations can also be represented as an augmented ancestor matrix $A(T)$, where every element is:

$$A_{i,k} = \begin{cases} 1, & \text{if } i = k \text{ or } i \text{ is an ancestor of } k, \\ 0, & \text{elsewhere.} \end{cases} \tag{2.2}$$

and every node is considered an ancestor of itself. For example, the augmented ancestor matrix for the tree shown in Figure 2.1c that we reduced to the mutation matrix E in Equation 2.1 is

$$\text{A} = \begin{array}{c} \\ M_1 \\ M_2 \\ M_3 \end{array} \begin{array}{c} \begin{array}{cccc} M_1 & M_2 & M_3 & R \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \tag{2.3}$$

Where each line and each column represents a different mutation M_1, M_2 and M_3 except for the fourth column that represents the root R of the mutation tree. The placement of the cells attached tot the mutation tree is represented by a vector σ, which holds at the j-th position the attachment point of sample j. Again relying on the example in Equation 2.1 and Figure 2.1c, we obtain $\sigma = (1, 1, 1, 4, 3, 3, 2)$ where 4 represents the root.

The connection between the mutation matrix E and the mutation tree represented by A is

$$(E_{ij}|T,\sigma) = A(T)_{i\sigma_j}. \tag{2.4}$$

In other words, for a given tree T and sample attachment σ, the mutation status E_{ij} of a sample is identical to the one observed in the node where the sample attaches to the tree.

2.1.2 Errors and missing data

In real situations though, we do not observe a perfect mutation matrix as in (Figure 2.1b). Usually we see a noisy version of it (Figure 2.1e) that we'll denote with D.
If the true mutation value is 0, we may observe a 1 with probability α (false positive), and if the true mutation value is 1, we may observe a 0 with probability β (false negative).
In presence of errors the likelihood of the data, given a mutation tree T, knowledge of the attachment of the samples σ and the error rates $\theta = (\alpha, \beta)$, is

$$\Pr(D|T,\sigma,\theta) = \prod_{i=1}^{n}\prod_{j=1}^{m} \Pr(D_{ij}|E_{ij}) \tag{2.5}$$

where E is the mutation matrix defined by T and σ, and n and m are respectively the number of detected mutations and the number of cells taken into exam.
Now using the posterior

$$\Pr(T,\sigma,\theta|D) \propto \Pr(D|T,\sigma,\theta)\Pr(T,\sigma,\theta) \tag{2.6}$$

we can factorize the prior, $\Pr(T,\sigma,\theta) = \Pr(\sigma|T,\theta)\Pr(T,\theta)$. We can also assume the independence of the error rates to set $\Pr(T,\sigma,\theta) = \Pr(\sigma|T)\Pr(T)\Pr(\theta)$. This way the attachment prior $\Pr(\sigma|T)$ will depend on T. Such a prior might be useful if one suspects that cells are more likely to be sampled from later stages in tumor development and lower down in the tree, although here we sample using a uniform distribution.

2.1.3 Markov Chain Monte Carlo

The model with which they try to learn the mutation histories from single-cell mutation profiles consists of three parts: the mutation tree T, the sample attachment vector σ , and the error rates of the sequencing experiment θ. The resulting search is too big to allow for an exhaustive search, so using Equation 2.5 and Equation 2.6 is possible to build a MCMC sampling scheme to sample from the joint posterior, given the data. The scheme moves from a state (T, σ, θ) to a new state (T, σ', θ') with an ergodic mixture of moves changing one component at a time. With properly defined transition probabilities and acceptance ratio, the chain will then converge to the posterior and so we'll be able to use the chain to approximate such posterior. In practice, however, it is possible to marginalize out the sample attachments σ in order to speed up convergence and focus on the mutation tree T as the informative part for understanding the mutation history.
This can be achieved by first noting that a move where we pick uniformly a sample and a new parent for that sample would satisfy the necessary properties for the MCMC chain on σ to converge. Moreover, since the likelihood in Equation 2.5 can be rewritten as

$$\Pr(D|T, \sigma, \theta) = \prod_{i=1}^{n} \prod_{j=1}^{m} \Pr(D_{ij}|A(T)_{i\sigma_j}) \tag{2.7}$$

the convergence can be achieved much faster. This is because (as said in subsection 2.1.2), for a given tree and sample attachment, the mutation status of a sample is identical to the one observed in the node where the sample attaches to the tree. This way the likelihood can be computed directly from T and σ. Written like this, the likelihood in Equation 2.7 can be factorized into a product for each sample to be attached. As long as the prior $\Pr(\sigma|T, \theta)$ can be factorized (so that the attachment for each sample is independent from the others), we can include the priors as in Equation 2.6 and efficiently sum Equation 2.7 over σ to finally marginalize it out:

$$\begin{aligned}\frac{\Pr(T,\theta|D)}{\Pr(T,\theta)} &\propto \sum_{\sigma}\prod_{j=1}^{m}\left[\prod_{i=1}^{n}\Pr(D_{ij}|A(T)_{i\sigma_j})\right]\Pr(\sigma_j|T,\theta) \\ &= \prod_{j=1}^{m}\sum_{\sigma_j=1}^{n+1}\left[\prod_{i=1}^{n}\Pr(D_{ij}|A(T)_{i\sigma_j})\right]\Pr(\sigma_j|T,\theta)\end{aligned} \tag{2.8}$$

Since Equation 2.8 can be efficiently computed and it is now possible to search over the $(n+1)^m$ times smaller space of trees T and error rates θ, the MCMC convergence can be achieved at higher speed.

With the attachments σ marginalized out, we can now consider only MCMC moves in the joint (T,θ) space. We can change one component at a time to propose a new pair (T',θ') with transition probabilities $q(T',\theta'|T,\theta)$ and accepting moves with the ratio

$$\rho = \min\left\{1, \frac{q(T,\theta|T',\theta')\Pr(T',\theta'|D)}{q(T',\theta'|T,\theta)\Pr(T,\theta|D)}\right\} \tag{2.9}$$

to sample proportionally to $\Pr(T,\theta|D)$. Once we have sampled a tree, we can easily sample each attachment independently following Equation 2.8.

Now fixing the error rates θ it is possible to build a scheme on rooted mutation trees as follows. Given a tree T, we find the neighborhood of all trees reachable with the MCMC move from T. Then we sample a tree T' from this neighborhood accordingly to a probability $q(T',\theta|T,\theta)$ and accept the move with the probability in Equation 2.9.

Once the chain converges this scheme would allow us to sample trees proportionally to $P(T,\theta|D)$, as long as the moves are reversible (that is, if the move from T to T' can be proposed with a non-zero probability, the reverse move from T' to T can also be proposed with a non-zero probability), irreducible (that is, there has to exist a sequence of moves that leads from any tree to any other tree), and aperiodical (which can be ensured by including the tree T in its neighborhood or adding a non-zero probability not to move).

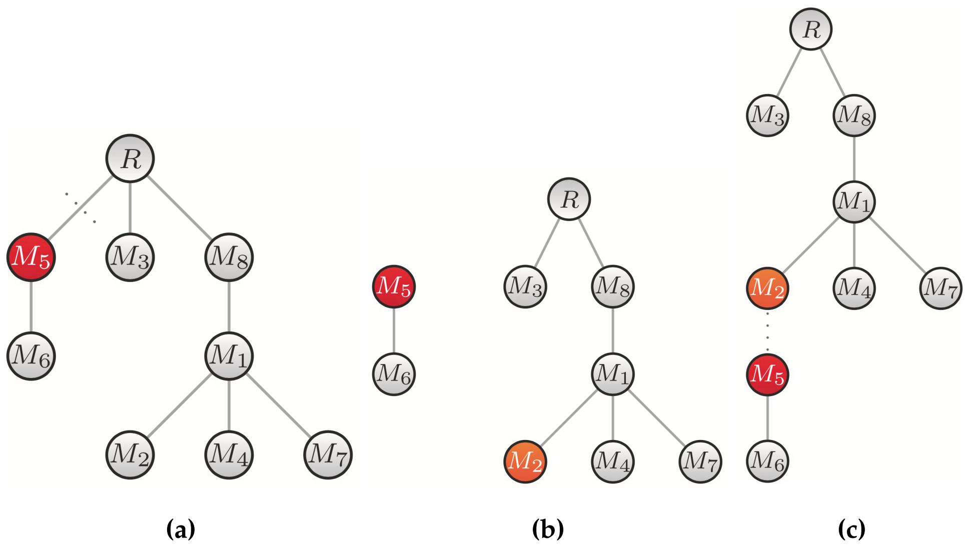


Figure 2.2. *Prune and reattach move.* ***(a)*** *Selection and detachment of the first node.* ***(b)*** *Selection of the second node.* ***(c)*** *Attachment of the first node on the second. Taken from [3].*

The MCMC move used is denoted *prune and reattach* (Figure 2.2). The first move consists on sampling a node i uniformly from the n available and cutting the edge leading to this node, removing its subtree from the tree, then the second move consists on uniformly sampling one of the remaining nodes (including the root) and the third move consists in attaching there the subtree.
The reverse move, in which we sample again i as first move but then pick its old parent as second move, has the same proposal probability $q(T, \theta | T', \theta) = q(T', \theta | T, \theta)$ since the set among which we chose the second node has the same size both times. The moves are also irreducible since there exist a path from any tree to a tree with all nodes attached to the root (obtained by moving each node to the root step by step). Finally they are also aperiodical since we can also choose the old parent when sampling a new one with non-zero probability. The *prune and reattach* move then suffices the conditions to sample trees according to their posterior. In order to speed up the convergence of the chain they implemented two other moves: *swap nodes* (in which they swap the labels of two nodes) and *swap subtrees* (in which

they swap two subtrees). At each step of the chain one of these three moves is selected with a fixed probability.

After convergence, the MCMC chain can be used along with Maximum Likelihood (ML) estimates. In this framework let's keep the error rates θ fixed for simplicity and consider the full space (T, σ) of trees with attachments. After maximizing over all possible placements, we can define the following score for each tree:

$$S(T) = \Pr(D|T, \sigma^*), \quad \sigma^* = \operatorname*{argmax}_{\sigma} \Pr(D|T, \sigma) \tag{2.10}$$

Since we can factorize the likelihood in Equation 2.7 it is possible to find the best attachment for each sample independently:

$$\sigma_j^* = \operatorname*{argmax}_{k} \prod_{i=1}^{n} \Pr(D_{ij}|A(T)_{ik}) \tag{2.11}$$

If more than one configuration provide the same maximum we can choose arbitrarily among them to calculate $S(T)$, that is:

$$S(T) = max_{\sigma} \Pr(D|T, \sigma) \tag{2.12}$$

with which we can find the maximum likelihood tree

$$T^* = \operatorname*{argmax}_{T} S(T) \tag{2.13}$$

Since the number of trees with $(n + 1)$ nodes (including the root) grows factorially, an exhaustive search becomes infeasible for trees with more than a few nodes. We can then use the MCMC scheme on this space of trees where, given a tree T, we propose a new tree T' according to one of the three move types explained above. The proposal probability $q(T'|T)$ remains the same but now we accept a move with probability

$$\rho = \min\left\{1, \frac{q(T|T')S(T')^{\gamma}}{q(T'|T)S(T)^{\gamma}}\right\} \tag{2.14}$$

γ is a parameter used to flatten the distribution (for $\gamma < 1$) or make it more pronounced (for $\gamma > 1$) depending on the type of search one is conducting.

In the Bayesian framework, we can also search for the MAP tree. With this scheme we would find the joint MAP tree and attachments by including the prior on all discrete components and accordingly updating $S(T)$ or we could search just for the MAP tree by instead averaging out the attachments.

2.1.4 SCITE Conclusions

In conclusion they say that SCITE's strength is the fact that it considers single cells as taxonomic units, meaning that we are working using a single cell as unit of measure, instead of a set of cells. The algorithm, when analyzing a cell, is capable of making use of the fact that each cell provides information about all the mutations, allowing for a more robust reconstruction of the mutation tree. This leads to a better identification of driver mutations, providing, however, a less certain placement of individual cells. On the other hand clustering cells into clones and considering these as the taxonomic units means that we can use the consensus of single-cell information in each clone to deduce more clearly the ancestral relationships between the clones themselves, but at the expense of reducing the accuracy in the reconstruction of the mutational history.
Even if, at the moment we are writing, the algorithm is not able to handle copy-number variations the authors claim that they will incorporate them into the model. One of the main challenges is the fact that in this case the infinite sites assumption wouldn't hold anymore.

2.2 Oncogenetic Nested Effects Model - OncoNEM

OncoNEM is a probabilistic method for inferring intra-tumor evolutionary lineage trees from somatic single nucleotide variants (SSNVs) of single cells.

The algorithm accounts for genotyping errors and tests for unobserved subpopulations, while clustering cells with similar mutation patterns into subpopulations and inferring relationships and genotypes of observed and unobserved subpopulations.

2.2.1 Likelihood of clonal trees

In OncoNEM we represent a binary genotype matrix of n cells each one with m possible mutation sites as $D = (d_{kl})$, where $k \in \{1, \ldots, n\}$ is the label of a single cell and $l \in \{1, \ldots, n\}$ is the index of a mutation site. $d_{kl} \in \{0, 1, \mathrm{NA}\}$ denotes then the mutation status (that could be respectively unmutated, mutated or unknown) of cell k at site l. This matrix together with the false positive rate (FPR) α and false negative rate (FNR) β consists in the input necessary to OncoNEM to infer the tumor subpopulations, a tree describing the evolutionary relationships among these subpopulations and the posterior probabilities of the occurrence of mutations.

In order to compute the likelihood associated with a clonal lineage tree T we assume it to be a directed, not necessarily binary, tree whose root is the unmutated normal (this is equivalent to restrict the search space of θ_l to $\{2, \ldots, N\}$). Each node of this tree represents a clone $c \in \{1, \ldots, N\}$ that contains 0, 1 or multiple cells of the data set. Now let $c(k)$ denote a clone containing cell k assuming, without loss of generality, that the root has index 1. Together with a lineage tree we also need the occurrence parameter $\Theta = \{\theta_l\}_{l=1}^{m}$, where θ_l is equal to the value c of the clone where mutation l appeared for the first time.
Given a dataset D we can derive the posterior probability of T and Θ given D as

$$\Pr(T, \Theta | D) = \frac{\Pr(D|T, \Theta) \Pr(\Theta|T) \Pr(T)}{\Pr(D)} \tag{2.15}$$

We assume the model prior $\Pr(T)$ (that can be used to incorporate prior biological knowledge) to be uniform over the search space. This together with the fact that the normalizing factor $P(D)$ is constant for all models and can be omitted, lets us rewrite Equation 2.15 as

$$\Pr(T, \Theta|D) \propto \Pr(D|T, \Theta) \Pr(\Theta|T) \tag{2.16}$$

Now we have two possible cases: either Θ is known or Θ is unknown.

If Θ is known we then know for each locus l in which clone the mutation occurred the first time. Let's also assume that no mutation occurred in the root. Then given a tree T and the occurrence parameter Θ we can predict the genotype of every cell. If c is the clone where a mutation occurred, then such mutation is present in c and all its descendants and is absent in all other clones. In other words given $\theta_l = c$, we can use the tree to determine the predicted genotype δ_{kl}.
To calculate the likelihood of (T, Θ), we compare the expected genotypes with the observed ones. We also model the genotyping procedure as draws of binary random variables ω_{kl} from the sample space $\Omega = \{0, 1\}$ and assume that, given T and Θ, the random variables are independent and identically distributed according to the probability distribution

$$\Pr(\omega_{kl}|\delta_{kl}) = \begin{pmatrix} \Pr(0|0) & \Pr(1|0) \\ \Pr(0|1) & \Pr(1|1) \end{pmatrix} = \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix} \tag{2.17}$$

where α and β are respectively the FPR and FNR respectively.
The observed genotypes d_{kl} are then interpreted as events from the event space $\mathcal{P}(\Omega) = \{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$ (a missing value NA corresponds to the event $\{0, 1\}$). Then the likelihood of observing the genotypes D given T and Θ is

$$\Pr(D|T, \Theta) = \prod_{l=1}^{m} \prod_{k=1}^{n} \Pr(\omega_{kl} \in d_{kl}|\delta_{kl}) \tag{2.18}$$

where

$$\Pr(\omega_{kl} \in d_{kl}|\delta_{kl}) = \begin{cases} 1-\alpha & \text{if } d_{kl} = \{0\} \wedge \delta_{kl} = 0 \\ \alpha & \text{if } d_{kl} = \{1\} \wedge \delta_{kl} = 0 \\ \beta & \text{if } d_{kl} = \{0\} \wedge \delta_{kl} = 1 \\ 1-\beta & \text{if } d_{kl} = \{1\} \wedge \delta_{kl} = 1 \\ 1 & \text{if } d_{kl} = \{0,1\} \end{cases} \tag{2.19}$$

is the probability of a single observation given the predicted genotype.

If Θ is unknown we can consider it as it was noise and marginalize over it. If we also make the assumptions that (1) the occurrence of a mutation in independent of the occurrence of all other mutations, such that

$$\Pr(\Theta|T) = \prod_{l=1}^{m} \Pr(\theta_l|T) \tag{2.20}$$

and (2) that the prior probability of a mutation occurring in a clone is

$$\Pr(\theta_l = c|T) = \begin{cases} 0/ & \text{if } c \text{ is the root (c = 1)} \\ \frac{1}{N-1} & \text{otherwise} \end{cases} \tag{2.21}$$

Then the marginal likelihood is

$$\begin{aligned} \Pr(D|T) &= \int \Pr(D|T,\Theta)\Pr(\Theta|T)d\Theta \\ &= \frac{1}{(N-1)^m} \prod_{l=1}^{m} \sum_{c=2}^{N} \prod_{k=1}^{n} \Pr(\omega_{kl} \in d_{kl}|T, \theta_l = c) \\ &= \frac{1}{(N-1)^m} \prod_{l=1}^{m} \sum_{c=2}^{N} \prod_{k=1}^{n} \Pr(\omega_{kl} \in d_{kl}|\delta_{kl}) \end{aligned} \tag{2.22}$$

2.2.2 Searching the tree space

With the scoring functions calculated in the previous section we can now search for the optimal tree.

OncoNEM inference is a three-step process where we start with an initial search, restricting the model space to cell lineage trees. Then we continue by testing whether adding unobserved clones to the tree might improve the likelihood. Finally we cluster the cells within the previously derived tree into clones to obtain the final tree.

2.2.2.1 Step 1: Initial search

Since the the search space of cell lineage trees with n nodes has n^{n-2} possible solutions, an exhaustive search by enumeration becomes infeasible for trees with more than nine nodes. Therefore they implemented function heuristicSearch to perform an heuristic local search. Note that the algorithm avoids getting stuck in a local optimum by returning to neighbors of high-scoring previous solutions.

2.2.2.2 Step 2: testing for unobserved clones

Since the number of sequenced single cells is usually small compared to the tumor size, some clones of the tumor may not be represented in the single cell sample. OncoNEM accounts for this possibility and tests if there is a lineage tree with new, unobserved, branch nodes that can better explain the observed data. Unfortunately unobserved clones that linearly connect observed clones cannot be inferred, but this is not a problem since they also do not change the shape of the tree.
In function expandTree we generate trees with $n+1$ nodes from a previous solution by inserting a new unobserved node into its branch points. These trees are used as start trees in a new heuristic search that will optimize the position of the new node node in the tree. Then a larger model is accepted if the Bayes factor of the larger versus the smaller model is bigger than a threshold ε. If the larger model passes the test, the steps are repeated, otherwise the algorithm terminates with the previous solution.

```
Input: Genotype matrix $D$, FPR, FNR, list $startTrees$ of starting trees (for the initial search
       the starting tree has a star topology), number of iterations $\delta$ without any
       improvement before stopping the execution
Output: List $consideredTrees$ of candidates scored trees
/* We define the neighbors of a given tree as those trees that can
   be generated from the current tree by assigning a new parent to
   one of the nodes or by swapping two nodes that are connected by
   an edge                                                        */
Initialize $consideredTrees$ as empty;
/* List ordered by likelihood                                     */
Initialize $priorityQueue$ as empty;
/* List of all trees that have been scored and whose neighbors have
   not yet been scored explicitly, ordered by likelihood          */
Initialize $counter = 0$ /* Counts search steps since last change of highest
   scoring solution                                               */
foreach $tree$ in $startTrees$ do
    score $tree$;
    add $tree$ to $consideredTrees$;
    add $tree$ to $priorityQueue$;
end foreach
$bestTree = consideredTrees[1]$;
while $counter \leq \delta$ do
    $currentTree = priorityQueue[1]$;
    delete $currentTree$ from $priorityQueue$;
    foreach $neighbor$ of $currentTree$ do
        if $neighbor \notin consideredTrees$ then
            score $neighbor$;
            add $neighbor$ to $consideredTrees$;
            add $neighbor$ to $priorityQueue$;
        end if
    end foreach
    if $bestTree \neq consideredTrees[1]$ then
        $counter = 0$;
        /* Highest scoring solution changed                       */
        $bestTree = consideredTrees[1]$;
    end if
    else
        $counter = counter + 1$;
    end if
end while
return $consideredTrees$
```

Function heuristicSearch

```
Input: Cell lineage tree $T_n$ with $n$ nodes inferred by function heuristicSearch, Bayes factor
        threshold $\varepsilon$
Output: Expanded tree $T_{n+i-1}$
1  Initialize $i = 0$;
2  repeat
3      $i = i + 1$;
       /* Generate start trees                                          */
4      $startTrees$ = star tree with $n + i$ nodes;
5      foreach node in $T_{n+i-1} thathasatleasttwochildren$ do
6          Generate a new tree by inserting an unobserved node into the branch point;
7          Add tree to $startTrees$;
8      end foreach
9      Pass $startTrees$ to function heuristicSearch and store the output in $consideredTrees$;
10     $T_{n+i}$ = highest scoring tree in $consideredTrees$ in which every unobserved node has at
        least two children;
       /* Calculate Bayes factor                                        */
11     $K = \Pr(D|T_{n+i}) / \Pr(D|T_{n+i-1})$;
12 until $K < \varepsilon$;
13 return $T_{n+i-1}$
```

Function expandTree

```
Input: Cell lineage tree $T$ inferred by expandTree(), Bayes factor threshold $\varepsilon$
Output: Tree $T$ clustered
/* Current tree                                                          */
Initialize $T = Tstart$;
/* Best tree scored so far                                               */
Initialize $T^* = Tstart$;
repeat
    foreach edge $e_i$ do
        Generate clustered tree $T_{e_i}$ from $T$ by merging the clones connected by $e_i$;
    end foreach
    $T_{e_i^*} = \operatorname{argmax}_{T_{e_i}} \Pr(D|T_{e_i})$;
    $K = \Pr(D|T^*)/\Pr(D|T_{e_i^*})$;
    if $K \leq \varepsilon$ then
        /* Accept clustering solution                                    */
        $T = T_{e_i^*}$;
        if $\Pr(D|T^*) < \Pr(D|T_{e_i^*})$ then
            /* Save clustering solution as new best tree                 */
            $T^* = T_{e_i^*}$;
        end if
    end if
until $K > \varepsilon$;
return $T$
```

Function clusterTree

2.2.2.3 Step 3: clustering cells into clones

There might be the possibility that the data could be better or equally well explained by a clonal lineage tree in which a single node might contain multiple cells (function clusterTree).
Nodes are iteratively merged along branches until the likelihood decreases by more than a factor $\frac{1}{\varepsilon}$ in respect to the best clustering solution found. The criteria for merging cells could be because they are genetically very similar or because the limited information content of the data makes us see those cells as possibly similar.

2.2.3 Bayes factor ε

The choice of the Bayes factor ε has to be done keeping in mind that it represents a trade-off between declaring clones with little support from the data and an overly strict clustering. In the paper's setting they chose $\varepsilon > 1$, meaning that they will prefer the smaller model unless the strength of evidence for the larger model compared to the smaller one exceeds a certain threshold. For a better understanding one can use as guide Jeffrey's [12] or Kass and Raftery's [13] scale for the interpretation of the Bayes factor. They report to have used a value of $\varepsilon = 10$, which denotes strong evidence according to Jeffreys's scale.

2.2.4 OncoNEM Conclusions

In conclusion they say that OncoNEM can be easily applied to present single-cell data sets. However, for larger data sets, the current search algorithm may become too computationally expensive. Unfortunately the model can't be used to model copy number variations, since they are not independent of each other and show horizontal dependencies [14], but the authors plan to extend the model to incorporate this kind of data in the future.
At the moment the the authors also recommend to to blacklist regions affected by

loss of heterozigosity (LOH) before applying OncoNEM inference, if additional data like bulk-sequencing is available, since OncoNEM is not fully capable of handling this kind of setting. So If is known that the evolution of the tumor is copy number driven and LOH affects very large parts of the genome, they recommend using a copy-number-based method instead for inferring tumor evolution.

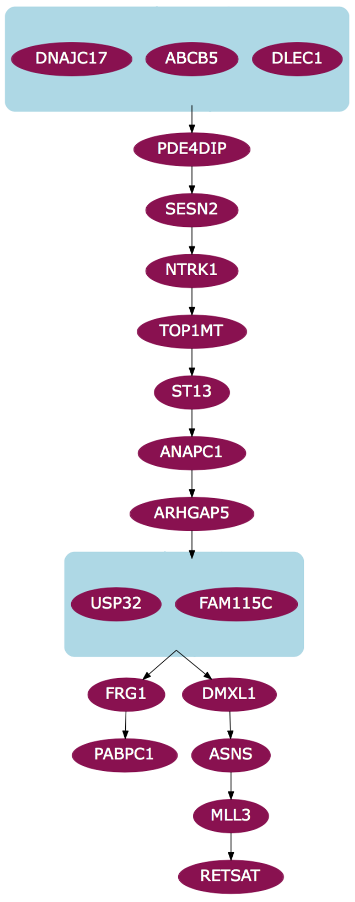
2.3 Comparison

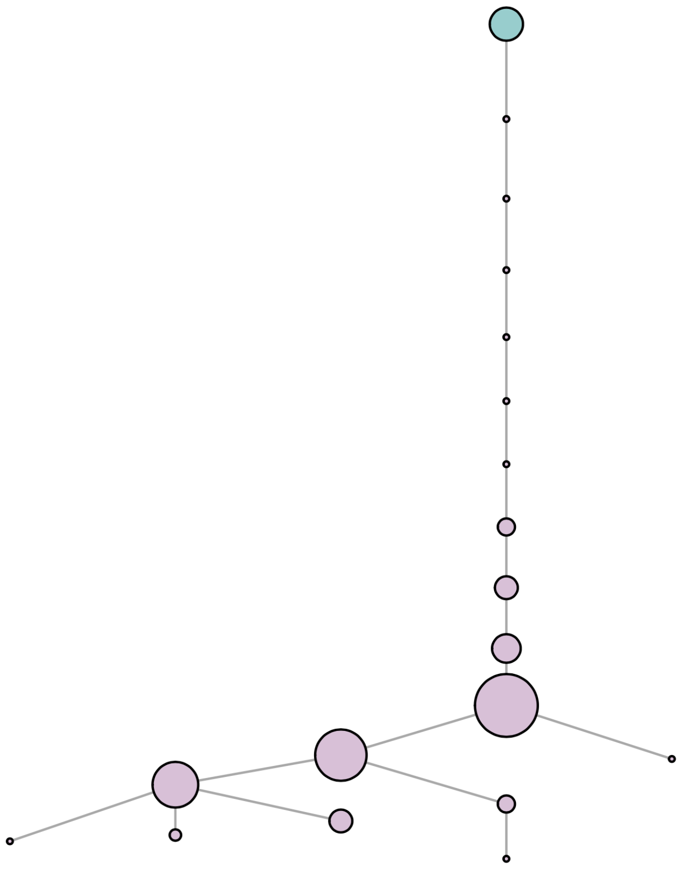
The two algorithms described in the previous sections are two (if not "the two") of the most recent algorithms used to reconstruct the phylogenetic history of a tumor.
Both of them make the same assumptions and base their scoring function on the likelihood function, what they differ is, to begin with, the fact that SCITE returns a mutation tree (where each node contains one or more mutations and the cells can be attached to the leaves of the tree) while OncoNEM returns an evolutionary lineage tree (in which every node contains one or more cells). Moreover OncoNEM performs an explicit clustering of the cells in its tree, while SCITE does something slightly different. Since it focuses on mutation rather than the single cells, the only kind of clustering involved in the computation is performed when is not possible to understand the ancestral relationships among two or more mutations. However one could see a sort of implicit clustering, similar to OncoNEM's, when we choose to attach the single cells to the respective leaves of the mutation tree.
While they both return their respective maximum-likelihood tree they implement two different approaches. SCITE uses an MCMC sampling method to build a chain which converges at its equilibrium to the posterior probability. It then uses MAP or ML estimates to obtain the the maximum-likelihood tree from this distribution. OncoNEM, on the other hand, implements a three-step algorithm considering only the likelihood score. After heuristically finding a set of candidate trees, for each one

of them, they try to improve its likelihood score first by adding possible missing nodes and then by trying to understand if a clustered tree might further improve the score.

Even though the two algorithms return two different kinds of trees we can show their results on a common dataset. Both of them have been tested on the dataset derived by a single-cell exome sequencing of 58 single cells from an essential thrombocythemia [8]. it is worth mentioning (although probably just a typo) that OncoNEM reports that Hou at al. estimated a FPR of 6.4×10^{-5} and a FNR of 0.42, while SCITE reports a FPR of 6.04×10^{-6} and a FNR of 0.4309. By looking directly at Hou et al. paper we saw that they actually estimated a FPR of 6.04×10^{-5} and a FNR of 0.4309.

The trees reconstructed by SCITE and OncoNEM are shown respectively in Figure 2.3a and Figure 2.3b. Even if they do not represent the same kind of tree we should expect some similarities. In fact we can see how they both share the same mostly linear structure with very few branches in the lower part of the tree. Also both algorithms performed better than any other similar algorithm they were compared with, obtaining log-likelihood scores way higher than the others. SCITE calculated a score of -378.4 against a best score of -1059.7 from its competitor. OncoNEM, on the other hand, obtained a score of -9964 compared to a -11584.





(a) (b)

Figure 2.3. *(a) SCITE's reconstructed tree from Hou et al. data. Picture taken from [3].* ***(b)*** *OncoNEM's reconstructed tree from Hou et al. data. Picture taken from [11].*

3 Simulator

This chapter describes the algorithm that has been developed. Section 3.1 introduces the general model, section 3.2 describes the algorithm and section 3.3 describes the code implementation.

3.1 Model

The program produces a phylogenetic tree describing the potential evolution of a tumor by simulating the duplication process of a cell that could lead to the creation of a tumoral mass.
The initial state is a single cell called *root*. When the simulation starts the cell begins its duplication process following a probability distribution (given as input). The duplication stops when the tree reaches a certain number of leaves. We selected this stopping condition (instead, for example, total number of nodes or a maximum height for the tree) in order to better simulate the progression over time of a given tumoral mass, since the number of cells in the tumoral mass, when it is extracted, is in fact the number of leaves of the phylogenetic tree that describes the evolution of such tumor.
By default the probability distribution that describes the duplication process of a

cell is given by two probabilities: p_0, that is the probability for a cell of not having any children (and so dying or entering a so called ″quescent″ state), set by default to 0.18 and p_1, the probability for a cell of having just one child, set by default to 0.45 (note that from these two is possible to derive the probability of having two children p_2 so there's no need to make it explicit). The value of p_0 has been obtained from the results of [15], in which they state that the death-birth rate δ of a cells population is usually in the range $[0.72, 0.99]$; this toghether with a standard birth rate of 0.25 (also according to [15]) make it possible to easily calculate that $p_0 \in [0.18, 0.2475]$. p_1's value instead is based purely on the intuition that probably it is more plausible for a cell to have a higher probability of generating one child than two children.

Every time a cell duplicates there's a chance that, due to many factors, one or more of the children might carry a mutation. We modeled this aspect with another probability distribution, of which we are interested in only two values: pd (the probability for a mutation to be *driver*) that has a default value of 0.00015 and pp (the probability for a mutation to be *passenger*) with a default value of 0.01485. We derived the values of these two probabilities from the product of the average mutation rate $u = 0.015$ of the exome part of the DNA of a single cell (as stated in [15]) and the probability $\Pr[\text{driver}|\text{mutated}] = 0.01$ (respectively $\Pr[\text{passenger}|\text{mutated}] = 0.99$), based on the assumption that on average only 1% of the exome mutations end up being drivers.

The effects on a cell's behavior of a *driver* mutation can be many: it could make the cell slightly more resistant to antibodies (making it slightly more difficult to kill), or it could make it more resistant to some particular substances (again increasing its potential lifespan), or it could increase the duplication rate of the cell and its children and so on. Anyway, in general, the long term effect of a *driver* mutation on a cell's behavior is to increase the number of offspring generated. Either by simply increasing the duplication rate or the cell's toughness, the final result is that the cell and its progeny will have an advantage over the others and will duplicate more easily. To model this fact we chose to decrease the value of p_0 of a value ε and to increase p_1 of $\varepsilon/2$ (effectively increasing also p_2 by $\varepsilon/2$) every time a *driver* mutation occurs. By default the value of ε has been set to 0.03.

After the simulator creates a tree the aim is to find a way to distinguish the potentially harmful mutations (drivers) and the harmless ones (passengers). In order to do this three statistical tests have been used: the Two Sample Kolmogorov-Smirnov Test, the 2 Sample Anderson-Darling Test and the Two Proportion Z Test. The Two Sample Kolmogorov-Smirnov Test and the 2 Sample Anderson-Darling Test have the purpose to find out if two samples come from the same probability distribution or not. The assumption is that when a *driver* mutation happens in a cell, the behavior of its subtree will deviate from the standard behavior that it would have had if the mutation was *passenger* or if the cell wouldn't have mutated at all. This means that if we consider a subtree as the output of a stochastic process, the stochastic process that generated a tree whose root is a *driver* mutated cell will be different from the one that generated a tree with a *passenger* mutated (or not-mutated) root. Of course this holds only in the case that the roots of these two trees have the same father, meaning that the two corresponding probability distributions will have the same past history and so that won't affect the analysis. If that's not the case then this reasoning can't be applied, since the different behavior of the two trees could be caused by the different past history of the two root cells. Now the problem becomes how to use the two tests to analyze these two subtrees since they take as input a set of numbers and not a tree. We chose to consider as output of the random variables associated with the stochastic process the number of cells in every subtree of every cell located at a certain depth from the root of the examined subtree. The idea is that since the duplication probabilities of a *driver* cell change in respect to the probabilities of a normal cell, the distribution of the number of cells present in those subtrees should be statistically different in the two trees if one of

the two roots has a *driver* mutation. The two tests are performed on every pair of siblings of which at least one is mutated, if the test returns a value that allows to reject the null hypothesis with sufficiently high probability then the mutated cell can be labeled as *potentially driver*. In the case both cells have a mutation nothing can be said. At the end of the analysis there are going to be two lists (one for each test) containing the cells on which it was possible to perform the tests and their associated corrected p-values. We say "corrected p-value" when referring to a p-value on which has been applied the Bonferroni correction [16] for multiple hypothesis testing. This correction allows to take into account the fact that since we are analyzing a large number of mutations (and so evaluating a large number of similar hypotheses), there's the possibility that what we observe is due simply to chance instead of a particular property of that cell.
The Two Proportion Z Test is a test used to assess if two proportions coming from different samples can be considered equal. The idea in this case is that if in a tree there are no *driver* mutations, the probability for a cell of not having children never changes. So if we consider a sample of cells at a certain depth (in such a way that it also corresponds to the state of a possible tumor at a certain time instant) we can estimate the probability of not having children of those cells by measuring the proportion of leaves in that sample. If the estimated probability can be deemed sufficiently similar to the default probability of not having any child then we can say with a certain degree of confidence that the subtree whose evolution led to that sample didn't came from a *driver* mutated root. This idea was applied to check if two subtrees, with the same constraints used for the tests explained above, could have been generated from the same type of cell or not. In order to do this we extracted two samples from the subtrees of two sibling cells, of which only one was mutated, and if the proportions of leaves in one of those samples could be considered statistically different from the other, then we would be able to say, with a certain degree of confidence, that the mutated sibling is in fact a *driver* mutated

cell.
In addition to this it is possible to perform another kind of analysis in order to simulate the kind of situation that can be found in currently available datasets. The aim is to model the fact that currently available datasets based on real data from single-cell sequencing techniques, come in fact from samples of a few hundreds of cells of bigger tumoral masses. The idea is that if we consider the leaves of the main tree as the whole tumor, then a sample of those leaves is indeed the sample of cells from which the real dataset has been derived. Once this subtree has been obtained then it is analyzed using the same method used for the main tree in order to identify the mutations that, with the highest probability, are the ones responsible for the tumor we observe.

3.2 Algorithm

```
Input: Number of leaves $N$, prob. $p_0$ of dying, prob. $p_1$ of having only one
       child, prob. $p_2$ of having two children, prob. $pd$ for a child to
       develop a *driver* mutation, prob. $pp$ for a child to develop a *passenger*
       mutation, $\varepsilon$
Output: Phylogeny tree $T$ having $N$ leaves starting from a cell with
        reproductive probabilities: $p_0$, $p_1$, $p_2$, $pd$ and $pp$. Every time a *driver*
        mutation occurs $p_1$ and $p_2$ are both increased by $\frac{\varepsilon}{2}$ and $p_0$ is
        decreased by $\varepsilon$
1  Create root cell with default reproductive parameters $p_0$, $p_1$, $p_2$, $pd$, $pp$ and
   $\varepsilon$, set it is reproduction time $t$ and label it as *leaf* ;
2  while $T$ has less than $N$ leaves and there is at least one leaf that can reproduce do
3      Select the leaf with the least reproduction time and take off the *leaf* label;
4      Generate the resulting (possibly mutated) offspring according to its
         reproductive probabilities and label it as *leaf*;
5      foreach newly generated child $c$ do
6          if $c$ has a driver mutation then
7              Set $c.p_0 = c.p_0 - \varepsilon$;
8              Set $c.p_1 = c.p_1 + \frac{\varepsilon}{2}$;
9              Set $c.p_2 = c.p_2 + \frac{\varepsilon}{2}$;
10         end if
11     end foreach
12 end while
13 if $T$ has less than $N$ leaves then
14     Delete $T$ and go to line 1;
15 end if
```

Function BuildTree

```
Input: Phylogeny tree $T$, test samples size lower bound $lb$
Output: Lists $listZT$, $listKS$ and $listAD$ containing every "analyzable"
        mutated cell with the corresponding corrected p-value obtained
        respectively by the Two Proportion Z Test, the Two Sample
        Kolmogorov-Smirnov Test and the 2 Sample Anderson-Darling
        Test
/* A mutated cell is considered "analyzable" if it has
   a sibling and it is possible to retrieve a
   sufficiently large sample for the statistical tests
   from both their subtrees                                  */
1 foreach cell $c$ in $T$ that could be considered "analyzable" do
2     Try to retrieve two samples of size at least $lb$ from the subtree of $c$ and
        its sibling's;
3     if it is not possible to obtain such samples then
4         Continue with the next cell;
5     end if
6     Perform the statistical tests on both samples and apply the Bonferroni
        correction [16] obtaining the corrected p-values p-valZT, p-valKS and
        p-valAD;
7     Append p-valZT, p-valKS and p-valAD respectively to the lists $listZT$,
        $listKS$ and $listAD$;
8 end foreach
```

Function AnalyzeWholeTree

```
Input: Phylogeny tree $T$, test samples size lower bound $lb$, number of
       leaves samples $k$, leaves samples size $n$
Output: $k$ couples $\{listZT_i, listKS_i, listAD_i\}$, $0 < i < k - 1$ of lists
        containing every "analyzable" mutated cell with the
        corresponding corrected p-value obtained respectively by the Two
        Proportion Z Test, the Two Sample Kolmogorov-Smirnov Test and
        the 2 Sample Anderson-Darling Test for each subtree
/* A mutated cell is considered "analyzable" if it has
   a sibling and it is possible to retrieve a
   sufficiently large sample for the statistical tests
   from both their subtrees                                  */
1  for $i = 0$ to $k - 1$ do
2      Extract a random sample $l$ of size $n$ from the set of leaves $L$ of tree $T$;
3      Extract the subtree $t$ that corresponds to the set $l$ of leaves (all the way
        up to $T$'s root);
4      foreach cell c in t that could be considered "analyzable" do
5          Try to retrieve two samples of size at least $lb$ from the subtree of $c$
            and its sibling's;
6          if it is not possible to obtain such samples then
7              Continue with the next cell;
8          end if
9          Perform the statistical tests on both samples and apply the
            Bonferroni correction [16] obtaining the corrected p-values p-valZT,
            p-valKS and p-valAD;
10         Append p-valZT, p-valKS and p-valAD respectively to the lists
            $listZT$, $listKS$ and $listAD$;
11     end foreach
12     Add $listZT_i$, $listKS_i$ and $listAD_i$ to the list of couples to be returned;
13 end for
```

Function AnalyzeTreeWithSamples

```
   Input: A different set of $\{N, p_0, p_1, p_2, pd, pp, \varepsilon, lb\}$ for each tree $T$ that you
          want to build and analyze in its entirety, or
          $\{N, p_0, p_1, p_2, pd, pp, \varepsilon, lb, k, n\}$ for each tree you want to analyze by
          sampling its leaves
   Output: A list of lists $mainList$ containing the output of
           function AnalyzeWholeTree and/or
           function AnalyzeTreeWithSamples for each set of parameters
           specified in input
1  foreach set of parameters $\{N, p_0, p_1, p_2, pd, pp, \varepsilon, lb\}$ given in input do
2      Pass parameters $\{N, p_0, p_1, p_2, pd, pp, \varepsilon\}$ to function BuildTree and
        retrieve its output $T$;
3      Pass $T$ and $lb$ to function AnalyzeWholeTree and retrieve its output
        $listZT_i$, $listKS$ and $listAD$;
4      Add $listZT_i$, $listKS$ and $listAD$ to $mainList$;
5  end foreach
6  foreach set of parameters $\{N, p_0, p_1, p_2, pd, pp, \varepsilon, lb, k, n\}$ given in input do
7      Pass parameters $\{N, p_0, p_1, p_2, pd, pp, \varepsilon\}$ to function BuildTree and
        retrieve its output $T$;
8      Pass $T$, $lb$, $k$ and $n$ to function AnalyzeTreeWithSamples and retrieve its
        set of $k$ output couples $s = \{listZT_i, listKS_i, listAD_i\}$, $0 < i < k - 1$;
9      Add $s$ to $mainList$;
10 end foreach
```

Algorithm 1: BuildAndAnalyze()

Algorithm 1 is the main procedure that takes care of building and analyzing the different trees that we want to generate according to the given input parameters. It starts by calling function BuildTree generating the tree T according to the first set of input parameters, then, depending on the kind of analysis that the user wants to perform, it calls either function AnalyzeWholeTree or function AnalyzeTreeWithSamples in order to analyze T and retrieve the results that are then stored in the list of lists $mainList$.
Function BuildTree is responsible to generate a new tree T according to the given input parameters. It starts by creating a single cell with the chosen reproductive parameters and basically it lets the cell free to reproduce until the associated phylogeny tree has the desired number of leaves. Since this is a random process it is possible that all the leaves die before their number reaches the desired threshold, in this case the faulted tree is destroyed and the process starts over until a proper tree has been generated.
Function AnalyzeWholeTree is responsible to analyze a tree T in order to find which are the mutations that most probably are *drivers*. As first thing the algorithm needs to know which mutated cells can be analyzed. Since the analysis process compares the behavior of two subtrees of two cells having the same father, only those cells that that are mutated and have a sibling can be considered to be "analyzable". Once all the mutated cells with siblings have been collected, for each one of them the algorithm tries to retrieve a sample (that's going to be used by the statistical tests) of size at least lb from them and their sibling, if that's not possible it continues with the next cell. If it is possible to retrieve such a sample then the mutated cell was indeed "analyzable" so the algorithm executes the statistical tests and stores the results in the corresponding output list.
Function AnalyzeTreeWithSamples instead performs a different analysis. In this case T is not analyzed in its entirety, instead the algorithm performs k random samples of its leaves of size n and for each sample extracts the corresponding

subtree (up to T's root). The same kind of analysis of the previous algorithm is then performed on each of these subtrees obtaining a set s of couples of lists.

3.3 Implementation

The main focus when implementing this algorithm was on scalability. The program needed to be able to generate statistically valid instances (whose size would allow the correct use of statistical tools), since the biggest datasets available today are made of a few more than 100 cells. So it was crucial to make the program as easy as possible towards the machine main memory. Because of this a massive use of pointers and references (to avoid making copies of data structures) was necessary.
it is important to mention that when talking about the size of an instance we refer to the number of leaves that the tree has (instead of the total number of nodes), since the leaves represent the stage of development, at a certain point in time, of an hypothetical potential tumoral mass.

The second most important focus was on speed. Working with such huge instances needed to be as fast as possible. For this reason the choice of the programming language fell on C++, that combines the convenience of an object oriented language with a speed that's just a bit lower than C's. Also the explicit use of pointers and references comes very useful when managing the impact of the program on the main memory is crucial.

Another important topic was about how to simulate the passing of time and be able to efficiently understand which will be the next cell that will duplicate. A priority queue ordered by the cell's duplication time seemed to be the best option. This queue contained pointers to the *active leaves* of the tree. A leaf is considered an *active leaf* if it hasn't done the duplication process yet.

Now a few details about the thread architecture of the program. In order to

be able to speed up the program's execution it has been implemented a simple multithreaded, and possibly multi-leveled, variant of the Producer-Consumer model in order to create multiple trees in parallel. The first stage of this multi-level model is formed by the *loader*, a single thread that has the job to read the parameters of the instances that need to be created and to store them in a thread-safe access queue *queue_L_S*. The second stage is formed by the *simulators*, a group of one or more threads with the purpose of building new instances according to the parameters they extract from *queue_L_S*. Once an instance has been built the thread passes it forward to the third stage using one of two thread-safe queues *queue_S_A1* and *queue_S_A2*, depending on the selected execution mode (explained below). The third stage is formed by the *analyzers*, a group of one or more threads that have the job to extract a completed instance from either *queue_S_A1* or *queue_S_A2* and analyze it in order to find the mutations that are more likely to be *drivers* using the kind of analysis that the user selected. The fourth and final stage is synchronized with the third by a third queue *queue_SW* and is formed by one or more threads called *sweepers* having the job to delete from the hard drive (if the selected execution mode so provides) the instances that have already been analyzed.
The fourth stage can be optionally activated and has been made because there is an execution mode in which the program does not store direct references or pointers to the newly created instances in the second queue but it stores the name of the text file where the instance representation can be found instead. Once an analyzer thread retrieves the name it rebuilds the instance in memory reading the file and proceeds to analyze it. This has been done in order to let the program be able to work on the same instances more than once, if needed, without the need to build new ones from scratch. The downside of this approach is, of course, the fact that there is a huge bottleneck on the writing and reading process on the hard drive, and due to the size of the files representing the instances this can become quite a problem on some systems.

And now a let's see the different execution modes.
There have been implemented a few global flags that trigger different execution modes in order to change the behavior of the program in resect to the user's needs. They are called: *activate_sweepers*, *one_at_a_time*, *save_hd_space*, *filename_simulator_inputs* and *work_with_samples*.
activate_sweepers is the flag responsible of the optional activation of the *sweepers* threads. When active these threads will help freeing memory on the hard drive by deleting the instances already analyzed.
one_at_a_time triggers a particular execution mode in which the multithreaded code executes as it was single-thread. Only one instance at a time is created and a new one is built only if the previous one has been completely analyzed. This mode hugely increases the execution time of the program, since all the benefits coming from a parallel execution are lost, but is easier on both the hard drive and the main memory, not mentioning the CPU.
save_hd_space has been implemented to fix the main bottleneck of the program's execution, that is the I/O operations involved in writing a new instance to a file and then read again that file to rebuild that instance in order to analyze it. When this flag is active the *simulators* will no longer write the new instances to file and pass its name to the *analyzers* using a queue *queue_S_A1*, but they will directly pass the new trees to the *analyzers* inside the main memory using another queue *queue_S_A2*. This way the execution speed can also be increased, provided that the machine on which the code is running has enough memory to contain the instances being created and analyzed. In any case this flag can be used together with *one_at_a_time* if the machine's main memory can't hold more than one instance at a time.
filename_simulator_inputs is not actually a binary flag, but is a string. By default it is set to the string "NULL", otherwise it contains the filename on which are stored the names of already built instances that the user wants to analyze again. If this is the case, no simulator thread will be activated since all the program has to do is

read the file and rebuild the instance from that.
work_with_samples triggers the execution of a different kind of analysis, in which the trees are not analyzed in their entirety but as described in the previous section.

Concerning the Two Sample Kolmogorov-Smirnov Test and 2 Sample Anderson-Darling Test used by the *analyzers* threads, since they weren't already implemented in C++, it was necessary either to implement them from scratch or use libraries imported from some other language. We chose the second option since the tests were too complex to try to implement them from scratch. In the end the choice fell onto the Python programming language since it was easier to embed in a C++ code, since a large part of it is written in C and the documentation and examples were sufficiently clear.
This does not mean it was an easy job. In particular the implementation of the garbage collection was a bit tricky. Python manages automatically the garbage collection by monitoring the reference count (the number of references associated with a certain object) of each object. Since C++ does not work this way there's the need to manually keep track of the reference count of each object that's been created and decrease or increase it in order not to create memory leaks or segmentation faults. The general rule is to decrease the reference counter once you are sure that you won't need that object anymore in the successive lines of code; references can also be manually increased but that's more rare and, if needed, it is usually told inside the API's documentation.
Another not so easy part to learn was how to use the Python C API in a multithreaded environment. By default the Python C API does not fully support multithreading and without the proper adjustments errors are very likely to occur, for example when two threads try to increase the same reference counter. In this case the API has implemented a global lock called *Global Interpreter Lock* (or GIL) that must be held by the current thread before it can safely access Python objects or call any kind of function from the API. The acquiring and releasing of the GIL

has to be done manually, however (in order to emulate concurrency) the Python interpreter tries to regularly switch threads, especially around potentially blocking I/O operations (like reading or writing a file) so that other Python threads can run in the meantime.

The code should run on most Unix based systems able to support at least the C++11 and Python 3.5 standards. In particular it was tested on a macOS 10.12.6 PC with an Intel Core i7 CPU 4770HQ having a clock speed of 2.2GHz and 4 physical cores with hyperthreading technology, 16GB of DDR3 RAM 1600MHz and a solid state hard drive running Python 3.6.2 and on the Linux based departmental cluster composed of 14 DELL PowerEdge M600 nodes with 2 quad core Intel Xeon E5450 with a clock speed of 3.00GHz, 16GB of RAM and 2 72GB hard drives in a RAID-1 configuration running Python 3.5.3.

4 Tools Used

4.1 Two Proportion Z Test

The Two Proportion Z Test is a statistical test used to determine whether the difference between two sample proportions p_1, p_2 is significant. The test procedure is appropriate when the following conditions are met:

1. The sampling method for each population is simple random sampling.

2. The samples are independent.

3. Each sample includes at least 10 successes and 10 failures.

4. Each population is at least 20 times as big as its sample.

As a side note we should mention that in some cases we executed this test even if not all these conditions were met. For example when analyzing a tree by sampling its leaves we need to use small samples of size not bigger than 10, since the trees we are analyzing are very small and so it could be difficult to extract bigger samples. So in these cases usually conditions 3 and 4 are not met.

For this test one can choose among three sets of hypotheses:

1. $p_1 - p_2 = 0$

2. $p_1 - p_2 \geq 0$

3. $p_1 - p_2 \leq 0$

In case 1 the test is a two tailed test, since an extreme value on either side of the sampling distribution would cause one to reject the null hypothesis. In cases 2 and 3 the test is a one tailed test, since an extreme value on only one side of the sampling distribution would cause one to reject the null hypothesis.

Once the *null hypothesis* has been stated it is possible to calculate the test statistic (*z-score*).
Let n_1 and n_2 be the sizes of the two samples associated respectively with p_1 and p_2, the *pooled sample proportion* P is:

$$\mathrm{P} = \frac{p_1 n_1 + p_2 n_2}{n_1 + n_2} \tag{4.1}$$

Given P we can compute the *standard error* S as:

$$\mathrm{S} = \sqrt{\mathrm{P}(1 - \mathrm{P})\left(\frac{1}{n_1} + \frac{1}{n_2}\right)} \tag{4.2}$$

Now we can compute the *z-score*.

$$z = \frac{p_1 - p_2}{\mathrm{S}} \tag{4.3}$$

Once we have the test statistic we can easily compute the corresponding p-value. Since the p-value is the probability of observing a sample statistic at least as extreme as the test statistic, that is a *z-score*, we can use a Normal Density Function to compute that probability depending on the kind of *null hypothesis* we chose at the beginning. If we performed a two tailed test then we'll have to consider both tails of the distribution, while if we chose a one tailed test we'll only need to consider one.

4.2 Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov Test is a non parametric, distribution-free test used to decide if a given sample could be generated by a specific random variable. The test statistic is based on the maximum distance between the empirical cumulative distribution function (*ECDF*) of the sample and the cumulative distribution function (*CDF*) of the reference distribution.

Given n iid observations the *ECDF* F_n is defined as:

$$F_n(x) = \frac{1}{n}\sum_{i=1}^{n} I_{[-\infty,x]}(X_i) \tag{4.4}$$

where $I_{[-\infty,x]}(X_i)$ is an indicator function equal to 1 if $X_i \leq x$ and equal to 0 otherwise.

Based on that the Kolmogorov-Smirnov statistic is defined as:

$$D_n = \sup_x |F_n(x) - F_0(x)| \tag{4.5}$$

where $\sup_x$ is the *supremum* of the set of distances between the *ECDF* $F_n(x)$ and the reference *CDF* $F_0(x)$.

By the Glivenko-Cantelli Theorem [17], if the sample comes from the reference distribution, then $D_n \xrightarrow{n\to\infty} 0$.

The strength of this test lays on the fact that under the *null hypothesis* H_0, that assumes that the empirical data comes indeed from the reference distribution, the Kolmogorov-Smirnov statistic D_n converges to a distribution (called Kolmogorov distribution) that does not depend on the empirical data.

In particular the Kolmogorov distribution is the distribution of the random variable

$$K = \sup_{t\in[0,1]} |B(t)| \tag{4.6}$$

whose cumulative distribution function is

$$\Pr(K \leq x) = 1 - 2\sum_{k=1}^{\infty}(-1)^{k-1}e^{-2k^2x^2} = \frac{\sqrt{2\pi}}{x}\sum_{k=1}^{\infty} e^{-(2k-1)^2\pi^2/(8x^2)} \tag{4.7}$$

where $B(t)$ is the Brownian Bridge [18].
Under the *null hypothesis* that the sample comes from the reference distribution $F_0(x)$ (and F_0 is continuous), $\sqrt{n}D_n$ converges to the Kolmogorov distribution (which does not depend on F_0)

$$\sqrt{n}D_n \xrightarrow{n\to\infty} \sup_t |B(F(t))| \tag{4.8}$$

In the *goodness-of-fit* test the *null hypothesis* is rejected at level α if $\sqrt{n}D_n > K_\alpha$, where K_α is calculated from $\Pr(K \leq K_\alpha) = 1 - \alpha$.

This test can also be used to assess whether two one-dimensional probability distributions differ. Given two samples of sizes n and m the Kolmogorov-Smirnov statistic becomes:

$$D_{n,m} = \sup_x |F_{1,n}(x) - F_{2,m}(x)| \tag{4.9}$$

where $F_{i,k}(x)$ is the *ECDF* of sample i of size k.
The *null hypothesis* that the two samples come from the same distribution is rejected at level α if

$$D_{n,m} > c(\alpha)\sqrt{\frac{n+m}{nm}} \tag{4.10}$$

The value of $c(\alpha)$ in general is equal to $c(\alpha) = \sqrt{-\frac{1}{2}\ln(\frac{\alpha}{2})}$. Below are reported the most common values of $c(\alpha)$:

α	0.10	0.05	0.025	0.01	0.005	0.001
$c(\alpha)$	1.22	1.36	1.48	1.63	1.73	1.95

Table 4.1. *Values of $C(\alpha)$ for the most common values of α*

Generally, though, what we consider when deciding whether to reject the *null hypothesis* or not is not directly the relationship between D_n and the significance level α, but the p-value corresponding to the observed value of the test statistic. In order to compute the p-value associated with the current D_n we first assume that the *null hypothesis* is valid. Under this assumption we know that the test statistic follows a predefined distribution called Kolmogorov Distribution. So, using this distribution, we can compute the probability to observe a value at least as "extreme" as the one we are observing at the moment (that is in fact the definition of p-value). If the obtained probability is not high enough it means that the assumed distribution didn't reflect the actual distribution of the data. We can then conclude, with enough confidence, that our initial hypothesis was in fact wrong, thereby rejecting the *null hypothesis*.

It is important to note that the Kolmogorov-Smirnov Test, as described up to now, can be used as long as the distributions involved are continuous. This does not mean that this test can't be applied to discrete distributions, as in the cases treated in this thesis, but the results have to be interpreted differently. In particular, in the case the distributions involved aren't continuous, the p-value returned by the test are conservative. So if p is the true p-value that is associated with the test between two discrete distributions $f_1(x)$ and $f_2(x)$, the observed p-value p' will be greater or equal than p. However this does not change the relative order between different p-values, meaning that if $p_1 < p_2$ then also $p'_1 < p'_2$.
This is why it was possible to use this test in this project, since the results are mostly based on the relative order between p-values and not their effective values.

4.3 Anderson-Darling Test

The Anderson-Darling Test [19] is a modification of the Kolmogorov-Smirnov Test that gives more weight to the tails of the distribution than the latter. While in its general form (so assuming that no parameters of the tested distribution need to be estimated) the Anderson-Darling Test is distribution free, like the Kolmogorov-Smirnov Test, the test is most often used in contexts when a family of distributions is being tested. In this case the computation of critical values and/or test statistics needs to take into account the tested distribution; of course this has the advantage

to allow a more sensitive test, but the disadvantage that the values have to be calculated for each reference distribution.
As the Kolmogorov-Smirnov Test, this test is based on the *CDF* (or *ECDF*) of the distributions taken into exam. The test statistic A to assess if a set of samples $\{Y_1, \dots, Y_n\}$, such that $Y_i \leq Y_{i+1} \ \forall \ 1 \leq i \leq n$, comes from a reference *CDF* F_0 is

$$A^2 = -n - S \tag{4.11}$$

where

$$S = \sum_{i=1}^{n} \frac{2i-1}{n} \left[\ln(F_0(Y_i)) + \ln\left(1 - F_0(Y_{n+1-i})\right)\right] \tag{4.12}$$

The critical values, in general, depend on the distribution being tested and the *null hypothesis* can be rejected if the statistic $A$is greater than a certain value.

This test can also be slightly modified in order to assess whether two or more samples can be considered outputs of the same random variable.
Given k different random samples with combined size and distinct values respectively n and $z_1, \dots, z_L$, such that $z_i \leq z_{i+1} \ \forall \ 1 \leq i \leq L$ (note that $L < n$ if there are tied observations), the test statistic becomes

$$A_k = \frac{n-1}{n^2(k-1)} \sum_{i=1}^{k} [\frac{1}{n_i} \sum_{j=1}^{L} h_j \frac{(nF_{ij} - n_i H_j)^2}{H_j(n - H_j) - n\frac{h_j}{4}}] \tag{4.13}$$

where h_j is the number of values in the combined samples equal to z_j, H_j is the number of values in the combined samples less than z_j plus $\frac{1}{2}h_j$ and F_{ij} is the number of values in the i-th group that are less than z_j plus one half the number of values in the same group that are equal to z_j. Also in this case the critical values usually depend on the data that's being tested and so they need to be calculated for each case.

Important to point out is the fact that, unlike the Kolmogorov-Smirnov Test, the Anderson-Darling Test can be used with every kind of distribution, be it continuous or discrete, without any issue.

5 Results

In the following chapter we report the results obtained by applying the statistical tests on different kinds of instances generated by the simulator algorithm that has been developed. Since, after implementing it, we found out that the 2 Sample Anderson-Darling Test had a lower minimum precision than the other two, we chose to exclude it from the results and consider only the Two Sample Kolmogorov-Smirnov Test and Two Proportion Z Test.

5.1 Experimental Design and Setup

In total we analyzed 51 different sets of instances, each one of them comprising instances with a specific common set of parameters. We chose to model 42 sets of instances so they would have only one *driver* mutated cell forced as one of root's children and 9 sets so they would have a single *driver* mutated cell but not forced in one particular position in the tree but let free to appear in any possible node. As for these 9 sets they differ by the size of the instances (intended as number of leaves of the trees) and by the lower bound on the size of the samples used by the statistical tests. Specifically we chose to consider instances of 1 million, 5 millions and 10 millions of leaves (mostly for machine-related compromises) and lower bounds of

$100, 200$ and 300. Looking at the other 42 instances, they were always divided in the same three sets of sizes mentioned before. For each one of them we changed the following parameters: (1) We first changed ε using the values $\{0.06, 0.1, 0.5, 0.9\}$ while keeping the statistical tests samples size lower bound equal to 200; (2) we then performed a sampled analysis and for each tree we extracted 100 samples of 100, 1000 and 10000 leaves, then the subtrees were analyzed keeping a fixed statistical tests lower bound of 2; (3) finally we kept every parameter with its default value except the statistical tests lower bound, that we changed using the values $\{2, 5, 10, 50, 100, 200, 300\}$.
For each one of these configurations we created and analyzed 500 different instances. This is true except for the ones where the trees were analyzed by sampling their leaves; in this cases were created 5 trees from which were extracted and analyzed 100 samples. So in total the number of subtrees analyzed is the same, but the runs are not all independent from each other.
The runs have been executed on the Linux based departmental cluster composed of 14 DELL PowerEdge M600 nodes, each with 2 quad core Intel Xeon E5450 with a clock speed of 3.00GHz, 16GB of RAM and 2 72GB hard drives in a RAID-1 configuration, running Python 3.5.3.

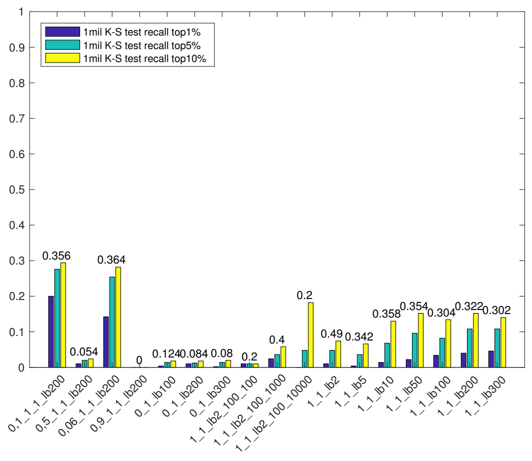
5.2 Two Sample Kolmogorov-Smirnov Test results

Applying the Two Sample Kolmogorov-Smirnov Test to the instances listed in the introduction of this chapter we obtained the results that we are about to report in this section.
We first grouped the instances by size, looking at the percentage of *driver* mutated cells (keeping in mind that each single instance is a tree with only one *driver* mutated cell) that the algorithm could rank in the top 5,10 and 50 positions and the percentage for *driver* mutated cells that the algorithm could rank in the top 1%,

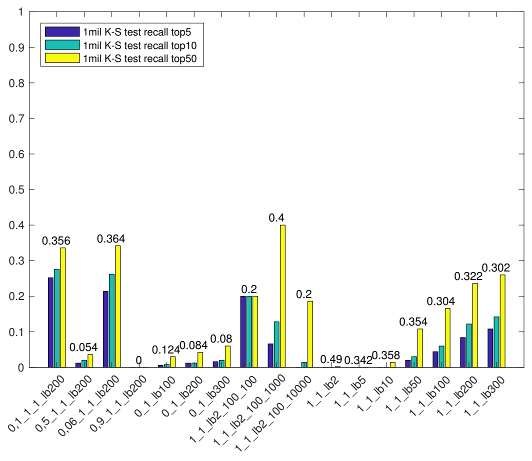
5% and 10%. The number on top of each bars triplet corresponds to the proportion of *driver* mutated cells that have been labeled ″analyzable″ and so the maximum value that those bars can achieve. For example, if we look at the bar labeled ″0,1_1_1_lb200″ in Figure 5.1a we can see that, out of 500 driver mutated cells (one for each instance) only 35.6% (178) were labeled ″analyzable″. Let's recall that a mutated cell can be labeled as ″analyzable″ when it has a sibling and is possible to retrieve from its subtree a sufficiently large sample to be used in the statistical tests.

We will first compare the results inside the four main groups highlighted before (that are: (1) *driver* mutation forced on root's child and varying ε, (2) *driver* mutation not forced in a particular position inside the tree, (3) *driver* mutation forced on root's child and analyze the trees by sampling their leaves and (4) *driver* mutation forced on root's child and varying the lower bound of the statistical tests) and then we'll compare the results among these four groups.

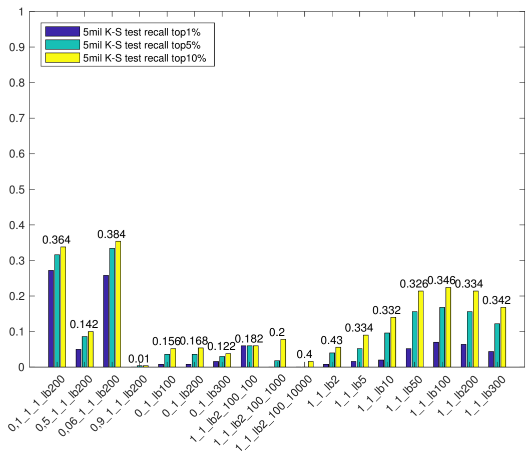
Let's begin by looking at Figure 5.1a and Figure 5.1b and considering the first group of four instances while keeping the bar labeled *1_1_lb200* as reference. This last bar corresponds to the base case in which ε is at its default value. We can see that increasing ε also increases the performances of the test, since the *driver* mutated cell appears more frequently in the ranking top positions. This is true up to a certain threshold, whose value is in the interval $(0.1, 0.5)$, where the performances begin to drop. This behavior, that might seem odd at first, is due to the fact that by increasing ε we proportionally decrease the probability p_0 of not having children for the offspring of the *driver* mutated cell. So if ε has a value greater than p_0 then every offspring of the *driver* mutated cell will have probability 0 of not generating any child. This implies that the *driver* mutated subtree will grow way faster, and then deeper, than the other one. For the kind of analysis we are performing this is not a favorable condition since for being able to analyze a cell we need that both that cell and its sibling have subtrees sufficiently deep to be able to retrieve



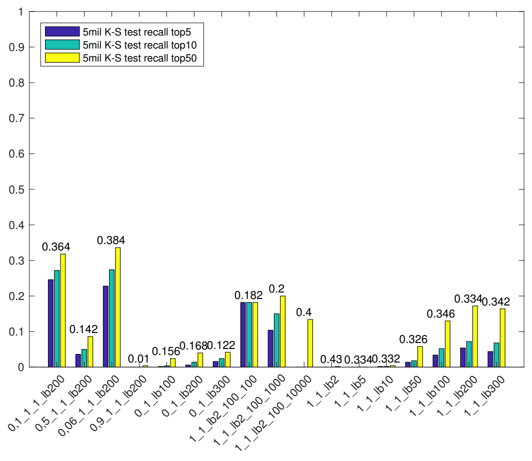
(a)



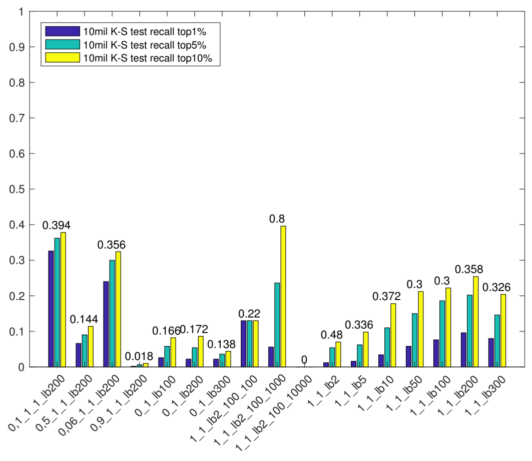
(b)



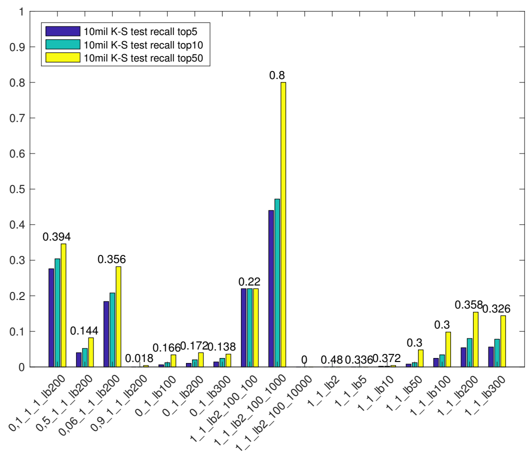
(c)



(d)



(e)



(f)

Figure 5.1. ***(a)*** *1 million size, top1%, 5% and 10%.* ***(b)*** *1 million size, top5, 10 and 50.* ***(c)*** *5 million size, top1%, 5% and 10%.* ***(d)*** *5 million size, top5, 10 and 50.* ***(e)*** *10 million size, top1%, 5% and 10%.* ***(f)*** *10 million size, top5, 10 and 50.*

a sample of cells at least of the desired size. In the case we are considering the *driver* subtree has probably monopolized the growth of the tree and didn't let its sibling 'subtree grow. So when we dive deep into the tree to retrieve the samples for the statistical tests we find that we can't extract a sufficiently large sample from the sibling 'subtree, making the driver mutated cell "not analyzable". This effect is visible in the instances where $\varepsilon = \{0.5, 0.9\}$, where the number of "analyzable" drivers is practically 0, while when ε is less than the default value of p_0 this problem does not occur. This behavior is consistent in both Figure 5.1a and Figure 5.1b.
Now let's consider the second group of instances, where the driver mutated cell wasn't forced to be a child of the root node. We can clearly see that this group of instances is the one with the lowest proportion of drivers both labeled "analyzable" and in the top positions. This highlights the fact that this kind of test, to be effective, needs the mutated cells to have a certain height in order to be able to retrieve a sample of cells of the desired size from its subtree.
Let's now look at the third group, where the instances were analyzed by sampling the leaves of the trees instead of looking at the entire tree. One thing that we can note by looking at Figure 5.1b is that almost every "analyzable" driver is ranked among the top50. This is because of the fact that with such little trees (these trees are very high but very narrow) the number of "analyzable" cells is very small (compared with the main tree) and so it happens that the number of "analyzable" cells is actually less than 50, so every the driver mutated cell will always appear int he top50. This is true for instances created with samples of 100 and 1000 leaves, but not for the bigger instances. In the latter case the fact that most of the driver mutated cells appear in the top50 is due to the increase in precision with the increase in instance size. In fact to really analyze the performance with this kind of instances we need to look at Figure 5.1a, where we can see that with the increase in size of the samples the algorithm manages to put more drivers in the top ranking positions. Another interesting fact can be noted by comparing these results with

the bar labeled "1_1_lb2". In both scenarios the same lower bound was used, but in one case the instance has been analyzed by sampling the leaves, while in the other we considered the tree as a whole. We can see that, despite the fact that the number of "analyzable" driver mutated cells is higher when analyzing the whole tree at once, the number of driver mutated cells in the top positions is higher when analyzing the tree by sampling.

And now the last group of instances, where the driver is still forced to be one of root's children and we only changed the lower bounds of the statistical tests. By looking at Figure 5.1a we can see that increasing the lower bound increases also the percentage of drivers in the top positions up to a certain value. When the lower bound value is higher than 50 the percentage of drivers both in the top positions and labeled "analyzable" stays mostly constant. This is true for the sizes we chose to analyze, but we actually do not expect this behavior to continue indefinitely. This is because by increasing the size of the samples for the statistical tests on one hand we improve the precision of the algorithm in discerning, while on the other hand we decrease the number of cells that can be considered "analyzable". So we actually expect to find a point after which the performances will begin to drop, because the gain in precision will be overruled by the loss in number of "analyzable" cells.

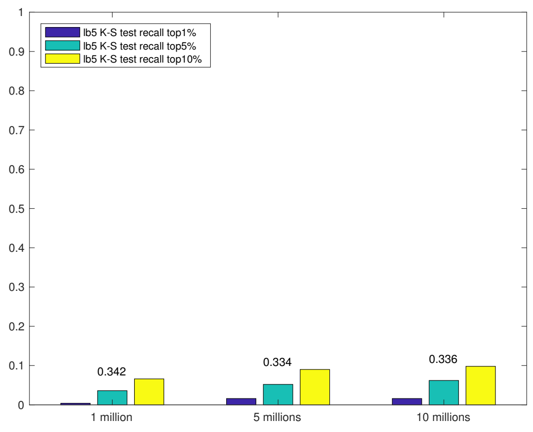
As a general comment we can say that the algorithm, for instances of size 1 million, is more sensible to variations of the value of ε (given that the variation is not too high) since the instances with the best average behavior in both Figure 5.1a and Figure 5.1b are the ones with $\varepsilon = \{0.03, 0.06\}$.

Let's now focus on Figure 5.1c and Figure 5.1d and follow the same steps as in the paragraph above. Most of what was said before for instances of size 1 million is valid also in this case. We need to mention though a few differences. First of all, by comparing Figure 5.1a and Figure 5.1c, we can see how the precision increases for almost every kind of instance, while by comparing Figure 5.1b and Figure 5.1d this

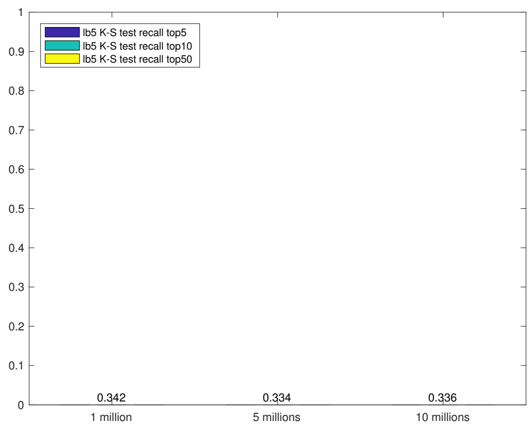
behavior is reversed. This is because we increased the size of the instances but kept fixed the thresholds to 5, 10 and 50 so with more mutations on the plate is simply more probable that the driver mutated cell does not appear in those top positions. The gain in performance can be seen by looking at the percentage thresholds where the increase in size of the instance was kept into account.
Another fact important to mention is that, looking at the instances of the last group of Figure 5.1c, now we can spot that descending trend that we expected when the loss in number of "analyzable" cells has overruled the gain in precision obtained with the increase of the size of the samples for the statistical tests. In fact we can clearly see that the maximum of that Gaussian-like curve is placed around 100 and after that value the general performances of the test begin to decrease.
This behavior can be better seen in the instances of size 10 million in Figure 5.1e and Figure 5.1f. In this case the maximum of the Gaussian-like curve moved to a bit higher value (this time around 200), as we expected since with an higher tree there's an higher chance to retrieve larger samples from an higher number of nodes. Also in this case we can see a general improvement in the performances by looking at the percentage top thresholds and a decrease by looking at the fixed top thresholds.

In Figure 5.2 we grouped the instances by the value of the lower bound used in the statistical tests.
The purpose is to show again how, by increasing the size of the instances analyzed or the value of the samples lower bound used in the statistical tests, we can improve the general performances of the algorithm and find a higher number of driver mutated cells in the top positions of our ranking. Also everything else we said, for example comparing the instances having the driver mutated cell forced in a particular position inside the tree with the instances where the driver mutated cell could be found anywhere in the tree, is confirmed by looking at these graphs.

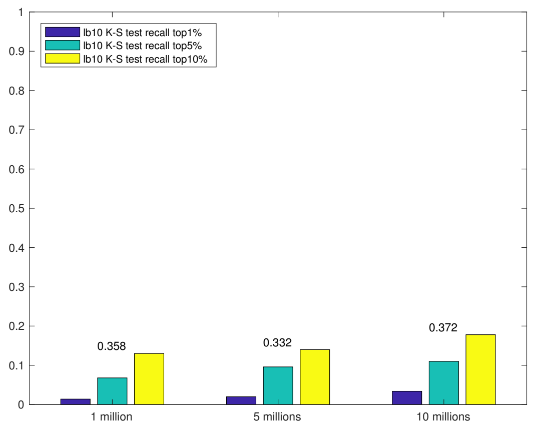
One other interesting performance evaluation could be done by analyzing the


lb5 K-S test recall top1%
lb5 K-S test recall top5%
lb5 K-S test recall top10%
0.342
0.334
0.336
1 million
5 millions
10 millions

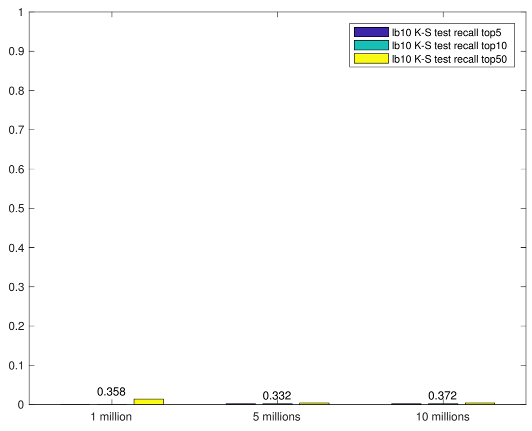
(a)


lb5 K-S test recall top5
lb5 K-S test recall top10
lb5 K-S test recall top50
0.342
0.334
0.336
1 million
5 millions
10 millions

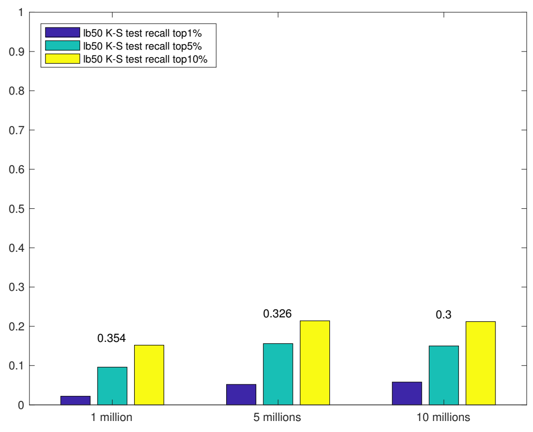
(b)


lb10 K-S test recall top1%
lb10 K-S test recall top5%
lb10 K-S test recall top10%
0.358
0.332
0.372
1 million
5 millions
10 millions

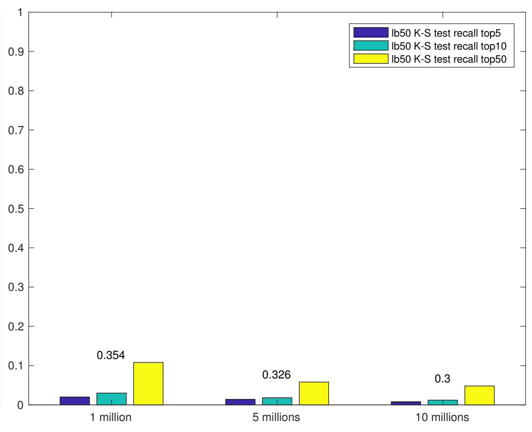
(c)


lb10 K-S test recall top5
lb10 K-S test recall top10
lb10 K-S test recall top50
0.358
0.332
0.372
1 million
5 millions
10 millions

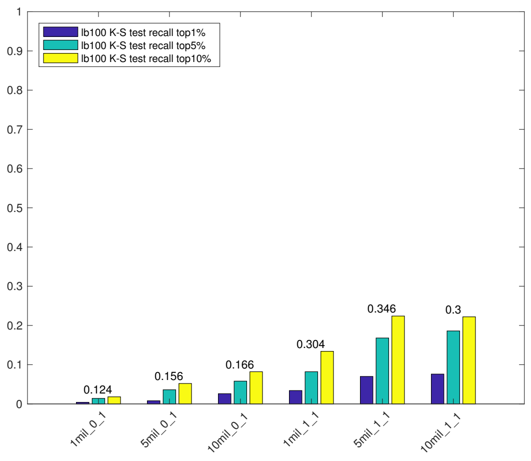
(d)


lb50 K-S test recall top1%
lb50 K-S test recall top5%
lb50 K-S test recall top10%
0.354
0.326
0.3
1 million
5 millions
10 millions

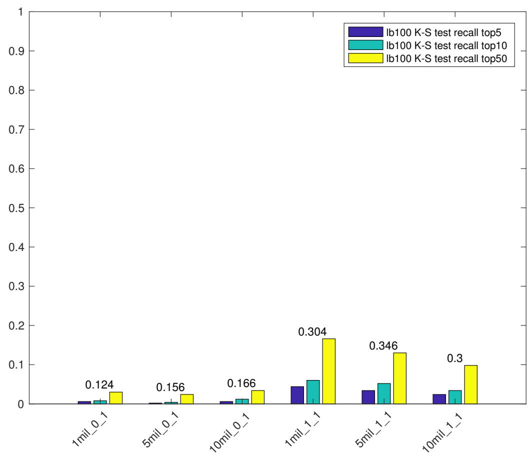
(e)


lb50 K-S test recall top5
lb50 K-S test recall top10
lb50 K-S test recall top50
0.354
0.326
0.3
1 million
5 millions
10 millions

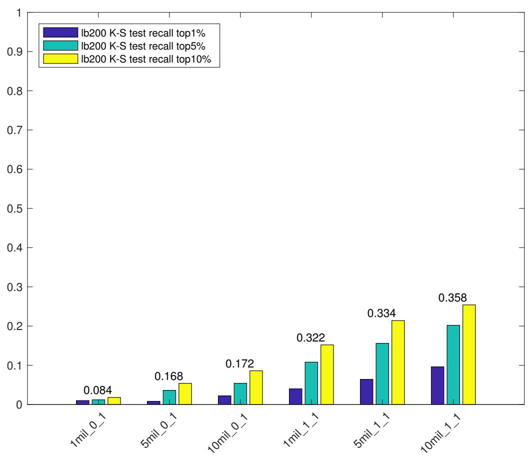
(f)



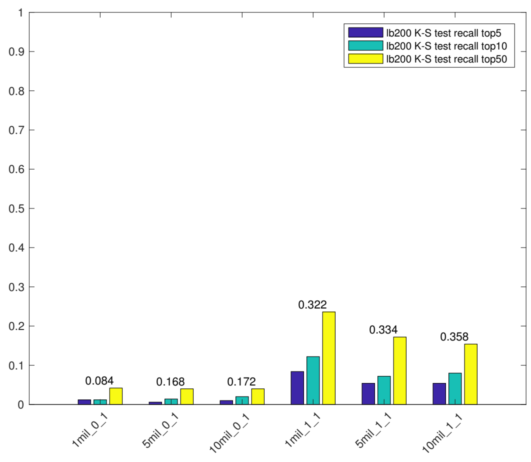
(g)



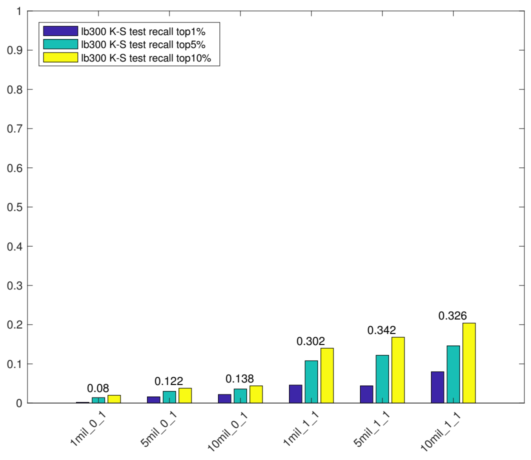
(h)



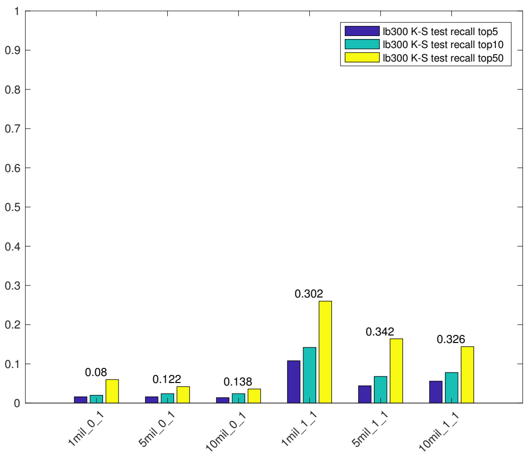
(i)



(j)

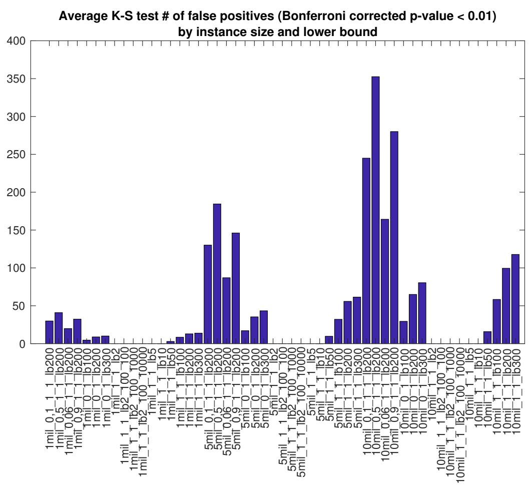


(k)

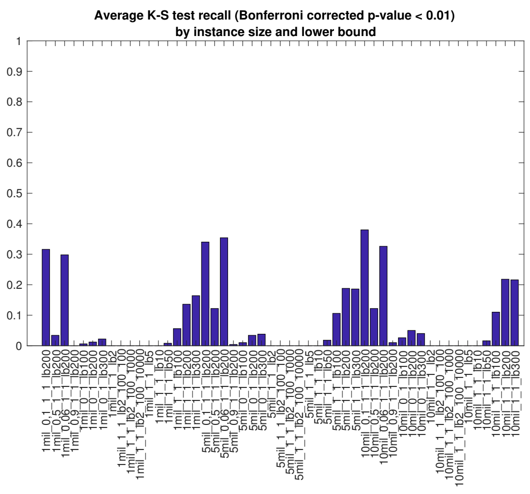


(l)

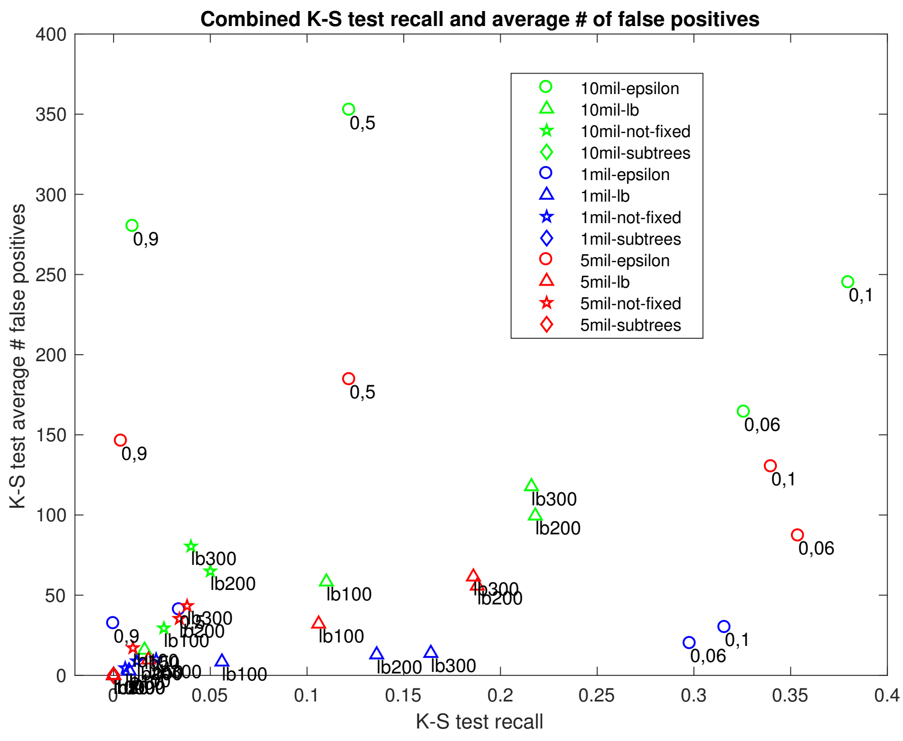
Figure 5.2. *__(a)__ Lower bound = 5, top1%, 5% and 10%. __(b)__ Lower bound = 5, top5, 10 and 50. __(c)__ Lower bound = 10, top1%, 5% and 10%. __(d)__ Lower bound = 10, top5, 10 and 50. __(e)__ Lower bound = 50, top1%, 5% and 10%. __(f)__ Lower bound = 50, top5, 10 and 50. __(g)__ Lower bound = 100, top1%, 5% and 10%. __(h)__ Lower bound = 100, top5, 10 and 50. __(i)__ Lower bound = 200, top1%, 5% and 10%. __(j)__ Lower bound = 200, top5, 10 and 50. __(k)__ Lower bound = 300, top1%, 5% and 10%. __(l)__ Lower bound = 300, top5, 10 and 50.*



(a)



(b)



(c)

Figure 5.3. ***(a)*** *Average number of false positives by instance.* ***(b)*** *Percentage of true positives by instance.* ***(c)*** *Combined visualization of graphs* ***(a)*** *and* ***(b)***.

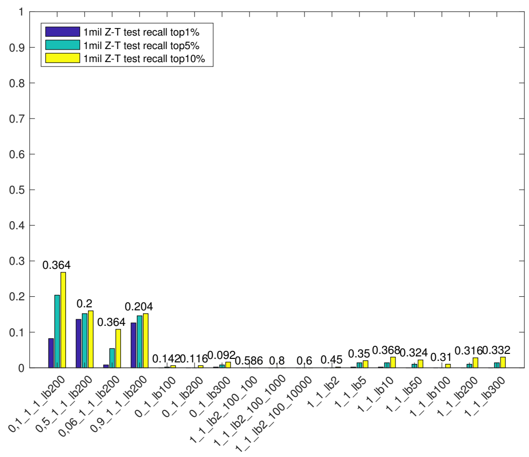
number of true and false positives that the algorithm manages to find for each kind of instance. We define a mutated cell as false positive if it is a passenger mutated cell and its associated p-value is less than 0.01, while we define a mutated cell as true positive if it is a driver mutated cell and its associated p-value is less than 0.01. In Figure 5.3a is reported the average number of false positives for each instance, while in Figure 5.3b is reported the proportion of driver mutated cells correctly identified (true positives). What we are looking for are instances with the highest ratio $\frac{true\ positives}{false\ positives}$. In order to better visualize such instances we combined Figure 5.3a and Figure 5.3b obtaining Figure 5.3c. In this figure we used the colors green, blue and red to highlight respectively instances with 10, 1 and 5 millions leaves and the shapes circle, triangle, star and diamond to distinguish respectively between instances where we varied the value of ε while keeping the driver mutated cell as root's child, instances where we varied the size of the statistical tests lower bound while keeping the driver mutated cell as root's child, instances where we kept every parameter with its default value but didn't force the driver mutated cell to be one of root's children and instances where we analyzed the trees not as a whole but by sampling their leaves. We then labeled every point with the corresponding value of the parameter that was modified.
Instances with the highest ratio $\frac{true\ positives}{false\ positives}$ are the ones placed in the lower right corner of the scatter plot. These instances are characterized by the highest percentage of true positives and the lowest average number of false positives. The instances with the highest ratio are then the ones where ε was the modified parameter and especially the ones having $\varepsilon = \{0.06, 0.1\}$. it is interesting to note that the better results are achieved with instances of 1 million leaves, while by increasing the size of such instances the performances seems to degrade. Actually we need to remind that all these instances have only one driver mutated cell at the top of the tree, so by increasing the size of such instances we also increase the probability of detecting false positives, since we are practically adding more

passenger mutated cells while keeping the driver one constant. So it is not surprising that the average number of false positives increases with the increase of the instances ′sizes. What we can see is also that the percentage of true positives increases with the instances ′size, proving again that bigger instances improve the algorithm precision. The second kind of instances that best suit our algorithm are the ones where the driver mutated cell is forced as a root′s child and the lower bound for the samples used in the statistical tests has the value 200 or 300. Also in this case we can see the same behavior we saw before, in which the percentage of true positives increases with the increase of the instance size together with the average number of false positives. The other kind of instances instead perform very poorly and so our algorithm is not suited to handle the kind pf parameters that characterize those instances.

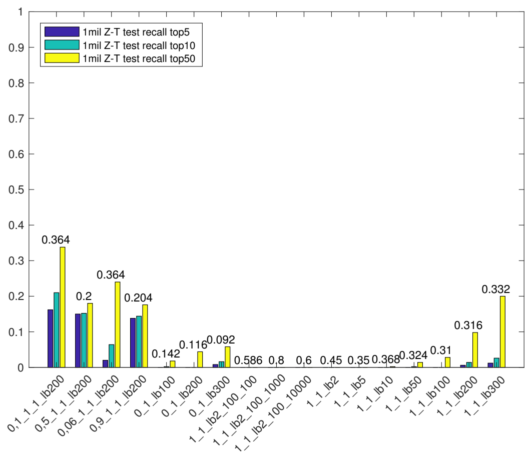
5.3 Two Proportion Z Test results

We also applied the Two Proportion Z Test to instances of the same kind as the ones listed before. Unfortunately, due to machine related and last minute issues, we were not able to perform this test on the exact same instances we used for the Two Sample Kolmogorov-Smirnov Test.
The analysis results were grouped as the ones in the previous section with the same criteria.

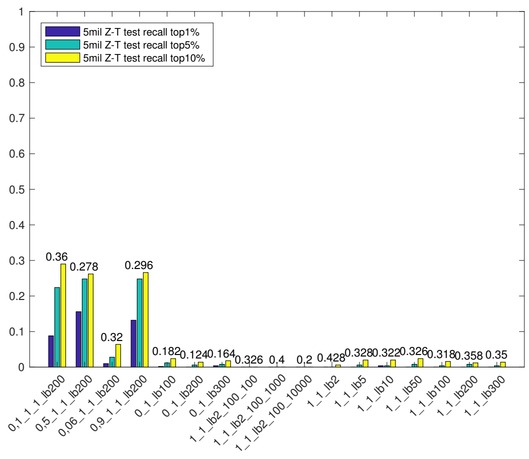
In Figure 5.4 instances are grouped by size.
This test, in terms of percentage of drivers found over the total of 500 (one driver per run), performs very poorly except in those scenarios where ε is the parameter we chose to change. While in these cases the algorithm has performances comparable with the Two Sample Kolmogorov-Smirnov Test (actually we can see that, in these cases, almost every ″analyzable″ driver is ranked in the top positions), in all other cases the algorithm is not capable of placing in the top positions not even



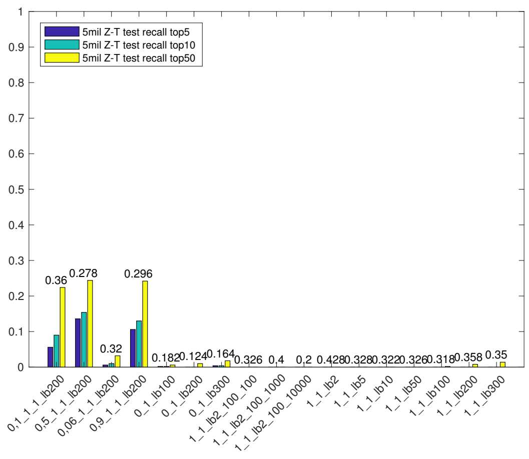
(a)



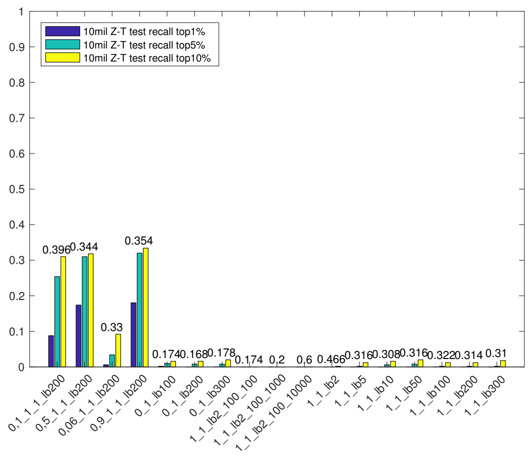
(b)



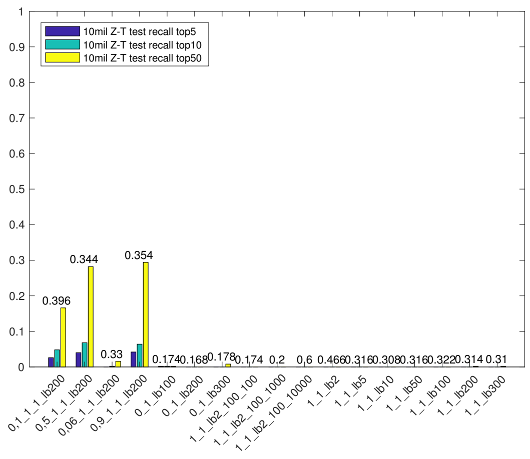
(c)



(d)



(e)



(f)

Figure 5.4. ***(a)*** *1 million size, top1%, 5% and 10%.* ***(b)*** *1 million size, top5, 10 and 50.* ***(c)*** *5 million size, top1%, 5% and 10%.* ***(d)*** *5 million size, top5, 10 and 50.* ***(e)*** *10 million size, top1%, 5% and 10%.* ***(f)*** *10 million size, top5, 10 and 50.*

half of the ”analyzable” drivers, even when increasing the size of the instances. This could indicate that the method used by this test has an intrinsic lesser precision than the one used by the Two Sample Kolmogorov-Smirnov Test. In fact even in the cases where the two tests didn’t analyze exactly the same instances the number of ”analyzable” driver mutated cells remains similar, but the number of such cells in the top ranks is much worse when looking at the Two Proportion Z Test.

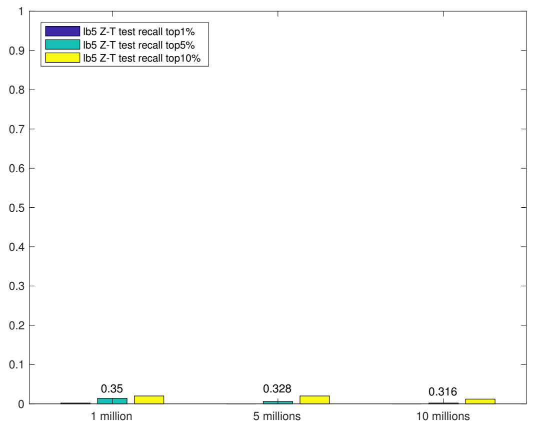
The poor performances are also visible in the bar graphs in Figure 5.5, where the results were grouped by the samples size lower bound used in the statistical tests.

For a better visualization of the performances of this test we can look at Figure 5.6c. We can see how instances with $\varepsilon = \{0.5, 0.9\}$ have almost a constant false positives, true positives rate varying the instances ’sizes, while the instances with $\varepsilon = 0.1$ don’t seem to notice the change in size. Surprisingly the instances with $\varepsilon = 0.06$ are way worse than these, while in Figure 5.4 it seemed that they had the same performances. This behavior is probably due to the fact that when $\varepsilon = 0.06$ the corrected p-values returned by the test are not high enough to allow to consider the associated cells as true or false positives, but if we look at their relative values then we can see that the test can still discriminate between the driver and not driver mutated cells well enough to place the first ones in the top ranking positions.

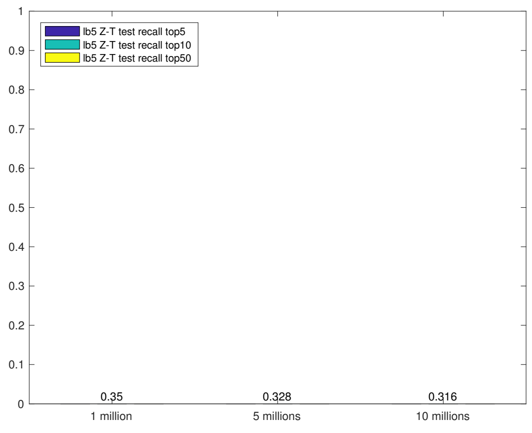
5.4 Combined comparison

It is interesting to show how the two tests perform when compared with each other on similar instances.

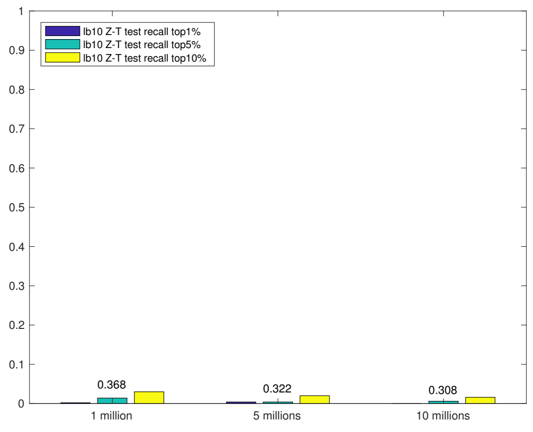
If we consider as performances measure the number of true and false positives and look at Figure 5.7 we can see that this test, even in the worst case, has a false positives rate lesser than the Two Sample Kolmogorov-Smirnov Test, while the recall can reach values comparable with the last test. This doesn’t change the


lb5 Z-T test recall top1%
lb5 Z-T test recall top5%
lb5 Z-T test recall top10%
0.35
0.328
0.316
1 million
5 millions
10 millions

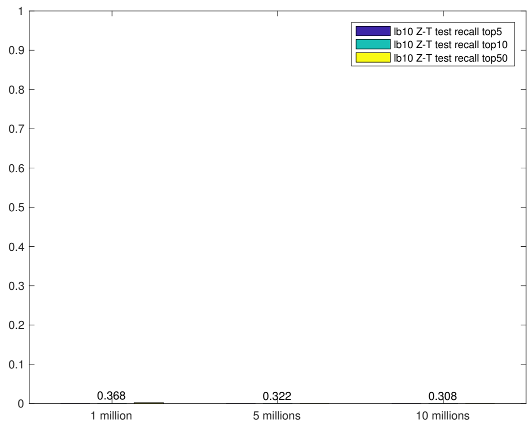
(a)


lb5 Z-T test recall top5
lb5 Z-T test recall top10
lb5 Z-T test recall top50
0.35
0.328
0.316
1 million
5 millions
10 millions

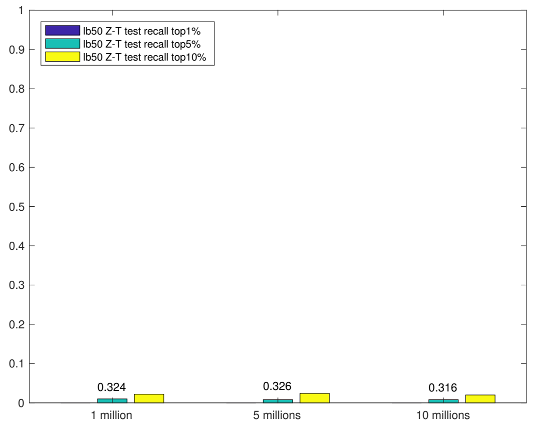
(b)


lb10 Z-T test recall top1%
lb10 Z-T test recall top5%
lb10 Z-T test recall top10%
0.368
0.322
0.308
1 million
5 millions
10 millions

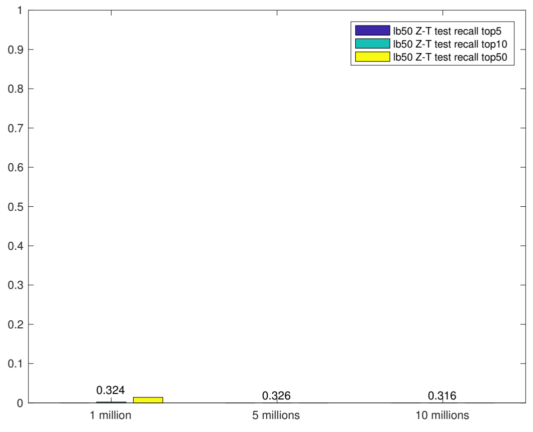
(c)


lb10 Z-T test recall top5
lb10 Z-T test recall top10
lb10 Z-T test recall top50
0.368
0.322
0.308
1 million
5 millions
10 millions

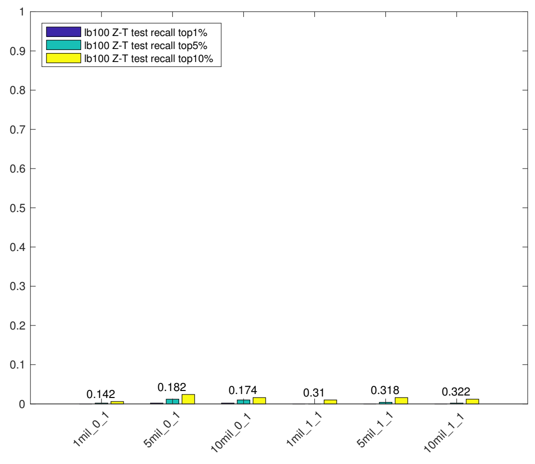
(d)


lb50 Z-T test recall top1%
lb50 Z-T test recall top5%
lb50 Z-T test recall top10%
0.324
0.326
0.316
1 million
5 millions
10 millions

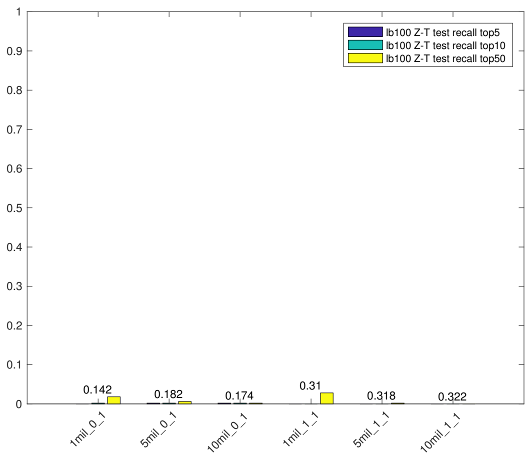
(e)


lb50 Z-T test recall top5
lb50 Z-T test recall top10
lb50 Z-T test recall top50
0.324
0.326
0.316
1 million
5 millions
10 millions

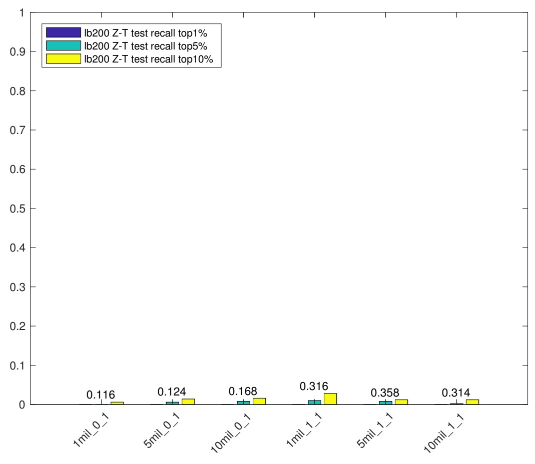
(f)



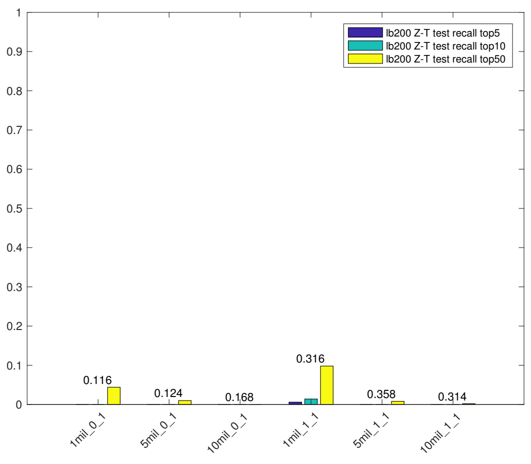
(g)



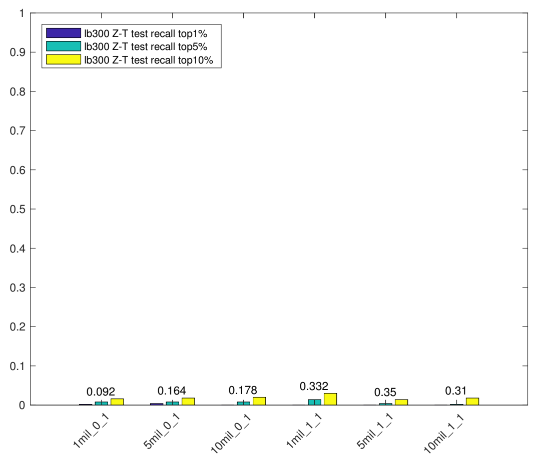
(h)



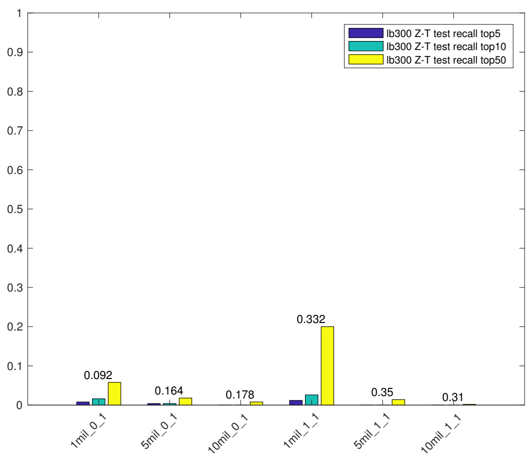
(i)



(j)

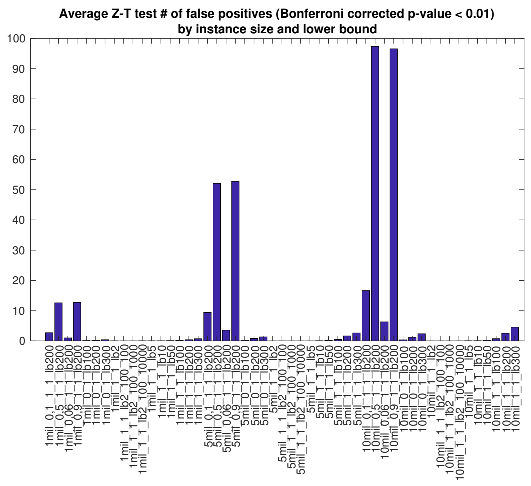


(k)

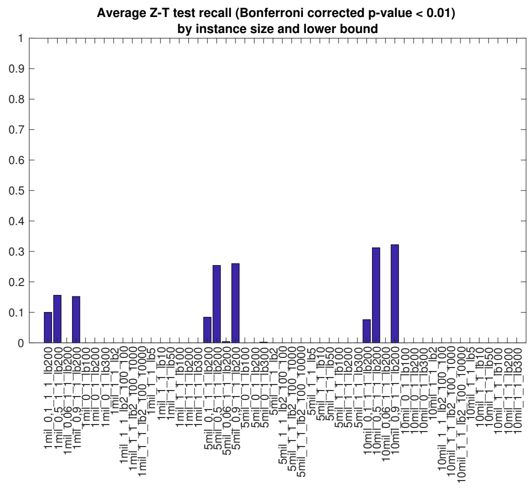


(l)

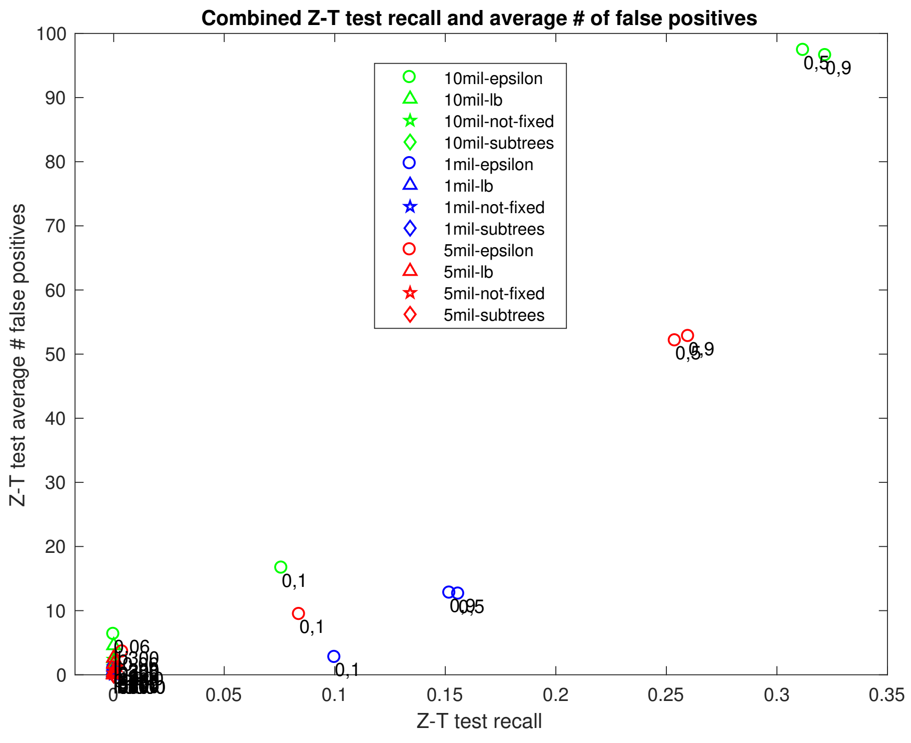
Figure 5.5. *(**a**) Lower bound = 5, top1%, 5% and 10%. (**b**) Lower bound = 5, top5, 10 and 50. (**c**) Lower bound = 10, top1%, 5% and 10%. (**d**) Lower bound = 10, top5, 10 and 50. (**e**) Lower bound = 50, top1%, 5% and 10%. (**f**) Lower bound = 50, top5, 10 and 50. (**g**) Lower bound = 100, top1%, 5% and 10%. (**h**) Lower bound = 100, top5, 10 and 50. (**i**) Lower bound = 200, top1%, 5% and 10%. (**j**) Lower bound = 200, top5, 10 and 50. (**k**) Lower bound = 300, top1%, 5% and 10%. (**l**) Lower bound = 300, top5, 10 and 50.*



(a)



(b)



(c)

Figure 5.6. ***(a)*** *Average number of false positives by instance.* ***(b)*** *Percentage of true positives by instance.* ***(c)*** *Combined visualization of graphs* ***(a)*** *and* ***(b)****.*

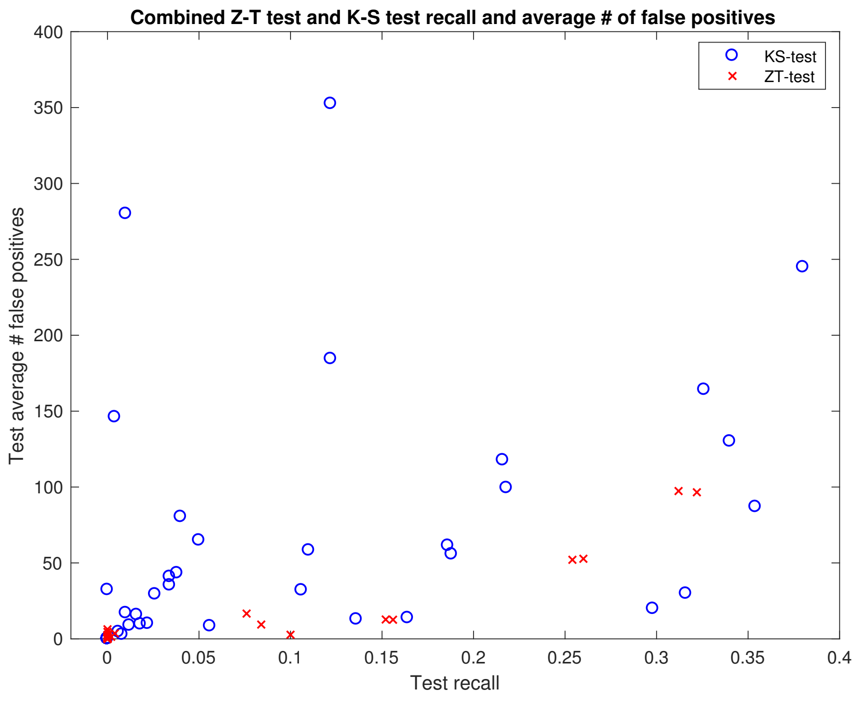


Figure 5.7. *Combined view of false and true positives for the Two Sample Kolmogorov-Smirnov Test and the Two Proportion Z Test*

fact that the Two Sample Kolmogorov-Smirnov Test, on average, performs better among all kinds of instances.

6 Discussion and conclusions

This chapter restates the objectives of this thesis (section 6.1), then we discuss the conclusions of our results (section 6.2) and finally we have a few words about future improvements (section 6.3).

6.1 Objectives

In this thesis we aimed to understand if it was possible to characterize the mutations that occur during the evolution of a tumoral mass. Besides we also wanted to implement a model that could simulate the growth of a potential tumoral mass by building its phylogenetic tree.

6.2 Conclusions

In this thesis we introduced in chapter 1 the problem of working with datasets of single-cell sequencing data and the problem of rebuilding the evolutionary tree that led to the observed tumoral mass. We then stated the objective of this thesis, that is: trying to understand if it is possible to characterize mutations once the phylogeny tree of the tumoral mass is known. In chapter 2 we then described and compared two of the state-of-the-art algorithms in the reconstruction of trees capable of describing the history of a group of cells. In chapter 3 we explained the model we used to simulate the growing process of a potential tumoral mass from a single cell and we then tested the goodness of some statistical tools (described in chapter 4) in characterizing the mutations that occurred inside the phylogenetic tree. The results of those tests have been reported and analyzed in chapter 5.

The phylogeny tree model, described in chapter 3, was intended to be used to simulate datasets containing an higher number of cells than currently available datasets have. On these instances we used three statistical tests in order to try to characterize the mutations arisen during the duplication process. We implemented the Two Proportion Z Test, the Two Sample Kolmogorov-Smirnov Test and the 2 Sample Anderson-Darling Test, described respectively in section 4.1, section 4.2 and section 4.3. Unfortunately the selected implementation of the 2 Sample Anderson-Darling Test had an intrinsic low precision that forced us not to use it in our final tests. So we compared the performances of the other two tests in chapter 5, observing that the Two Proportion Z Test had lower performances than the Two Sample Kolmogorov-Smirnov Test. This was rather unexpected, since we believed the Two Proportion Z Test would have performed better than the Two Sample Kolmogorov-Smirnov Test. If the probability of not having children p_0 is constant among all cells, if we collect every cell at a certain level l, we expect the number of childless cells to be, on average, equal to the initial probability p_0. Probably what we miscalculated was the fact that we need cells samples of greater size than the ones we used or, instead of confronting two proportions, we need to confront two average proportions, averaged on different levels of the tree.

6.3 Future Work

Several areas of this thesis work can be extended and improved in a future work.

First of all there's the coding part. Since I can't call myself an expert nor in C++ nor in software engineering I'm sure that the code can be implemented in a much more efficient way, speaking about both coding style and general performances.
For sure some better results can also be achieved by using some other statistical tool that can better adapt to the specifics of the problem, since in our case we used some general statistical tests that do not take advantages on any of the specifics of this particular problem. In particular, a first step might be to upgrade the Two Proportion Z Test with the suggestion reported in section 6.2. We might improve the Two Proportion Z Test by considering not only the proportion on one level, but by averaging it on multiple levels. Of course we should take into account the fact that now the samples that we want to average are not independent anymore and probably a simple arithmetic mean is not correct.

Bibliography

[1] P. C. Nowell, “The clonal evolution of tumor cell populations,” *Science*, vol. 194, no. 4260, pp. 23–28, 1976. [Online]. Available: http://www.jstor.org/stable/1742535

[2] L. Yates and P. J Campbell, “Evolution of the cancer genome,” vol. 13, pp. 795–806, 10 2012.

[3] K. Jahn, J. Kuipers, and N. Beerenwinkel, “Tree inference for single-cell data,” *Genome Biology*, vol. 17, no. 1, p. 86, May 2016. [Online]. Available: https://doi.org/10.1186/s13059-016-0936-x

[4] M. Greaves and C. C. Maley, “Clonal evolution in cancer,” *Nature*, vol. 481, no. 7381, pp. 306–313, 01 2012. [Online]. Available: http://dx.doi.org/10.1038/nature10762

[5] M. R. Stratton, P. J. Campbell, and P. A. Futreal, “The cancer genome,” *Nature*, vol. 458, no. 7239, pp. 719–724, 04 2009. [Online]. Available: http://dx.doi.org/10.1038/nature07943

[6] C. Swanton, “Intratumour heterogeneity: Evolution through space and time,” *Cancer research*, vol. 72, no. 19, pp. 4875–4882, 10 2012. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3712191/

[7] X. Xu, Y. Hou, X. Yin, L. Bao, A. Tang, L. Song, F. Li, S. Tsang, K. Wu, H. Wu, W. He, L. Zeng, M. Xing, R. Wu, H. Jiang, X. Liu, D. Cao, G. Guo, X. Hu, Y. Gui, Z. Li, W. Xie, X. Sun, M. Shi, Z. Cai, B. Wang, M. Zhong, J. Li, Z. Lu, N. Gu, X. Zhang, L. Goodman, L. Bolund, J. Wang, H. Yang, K. Kristiansen, M. Dean, Y. Li, and J. Wang, “Single-cell exome sequencing reveals single-nucleotide mutation characteristics of a kidney tumor,” *Cell*, vol. 148, no. 5, pp. 886–895, Mar. 2012.

[8] Y. Hou, L. Song, P. Zhu, B. Zhang, Y. Tao, X. Xu, F. Li, K. Wu, J. Liang, D. Shao, H. Wu, X. Ye, C. Ye, R. Wu, M. Jian, Y. Chen, W. Xie, R. Zhang, L. Chen, X. Liu, X. Yao, H. Zheng, C. Yu, Q. Li, Z. Gong, M. Mao, X. Yang, L. Yang, J. Li, W. Wang, Z. Lu, N. Gu, G. Laurie, L. Bolund, K. Kristiansen, J. Wang, H. Yang, Y. Li, X. Zhang, and J. Wang, “Single-cell exome sequencing and monoclonal evolution of a jak2-negative myeloproliferative neoplasm,” *Cell*, vol. 148, no. 5, pp. 873 – 885, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0092867412002280

[9] Y. Li, X. Xu, L. Song, Y. Hou, Z. Li, S. Tsang, F. Li, K. M. Im, K. Wu, H. Wu, X. Ye, G. Li, L. Wang, B. Zhang, J. Liang, W. Xie, R. Wu, H. Jiang, X. Liu, C. Yu, H. Zheng, M. Jian, L. Nie, L. Wan, M. Shi, X. Sun, A. Tang, G. Guo, Y. Gui, Z. Cai, J. Li, W. Wang, Z. Lu, X. Zhang, L. Bolund, K. Kristiansen, J. Wang, H. Yang, M. Dean, and J. Wang, “Single-cell sequencing analysis characterizes common and cell-lineage-specific mutations in a muscle-invasive bladder cancer,” *GigaScience*, vol. 1, no. 1, p. 12, Aug 2012. [Online]. Available: https://doi.org/10.1186/2047-217X-1-12

[10] J. G. Lohr, V. A. Adalsteinsson, K. Cibulskis, A. D. Choudhury, M. Rosenberg, P. Cruz-Gordillo, J. M. Francis, C.-Z. Zhang, A. K. Shalek, R. Satija, J. J. Trombetta, D. Lu, N. Tallapragada, N. Tahirova, S. Kim, B. Blumenstiel, C. Sougnez, A. Lowe, B. Wong, D. Auclair, E. M. Van Allen, M. Nakabayashi, R. T. Lis, G.-S. M. Lee, T. Li, M. S. Chabot, A. Ly, M.-E. Taplin, T. E. Clancy, M. Loda, A. Regev, M. Meyerson, W. C. Hahn, P. W. Kantoff, T. R. Golub, G. Getz, J. S. Boehm, and J. C. Love, "Whole-exome sequencing of circulating tumor cells provides a window into metastatic prostate cancer," *Nat Biotech*, vol. 32, no. 5, pp. 479–484, 05 2014. [Online]. Available: http://dx.doi.org/10.1038/nbt.2892

[11] E. M. Ross and F. Markowetz, "Onconem: inferring tumor evolution from single-cell sequencing data," *Genome Biology*, vol. 17, no. 1, p. 69, Apr 2016. [Online]. Available: https://doi.org/10.1186/s13059-016-0929-9

[12] H. Jeffreys, *Theory of Probability*, 3rd ed. Oxford, England: Oxford, 1961.

[13] R. E. Kass and A. E. Raftery, "Bayes factors," *Journal of the American Statistical Association*, vol. 90, pp. 773–795, 1995.

[14] R. F. Schwarz, A. Trinh, B. Sipos, J. D. Brenton, N. Goldman, and F. Markowetz, "Phylogenetic quantification of intra-tumour heterogeneity," *PLOS Computational Biology*, vol. 10, no. 4, pp. 1–11, 04 2014. [Online]. Available: https://doi.org/10.1371/journal.pcbi.1003535

[15] I. Bozic, J. M. Gerold, and M. A. Nowak, "Quantifying clonal and subclonal passenger mutations in cancer evolution," *PLoS Computational Biology*, vol. 12, no. 2, 2016. [Online]. Available: http://dx.doi.org/10.1371/journal.pcbi.1004731

[16] (2017) Bonferroni correction. [Online]. Available: https://en.wikipedia.org/wiki/Bonferroni_correction

[17] (2017) Glivenko-cantelli theorem. [Online]. Available: https://en.wikipedia.org/wiki/Glivenko%E2%80%93Cantelli_theorem

[18] W. C. Chow, "Brownian bridge," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 1, no. 3, pp. 325–332, 2009. [Online]. Available: http://dx.doi.org/10.1002/wics.38

[19] (2017) Anderson-darling test. [Online]. Available: http://www.itl.nist.gov/div898/handbook/eda/section3/eda35e.htm

[20] (2017) Kolmogorov–smirnov test. [Online]. Available: https://en.wikipedia.org/wiki/Kolmogorov%E2%80%93Smirnov_test

[21] (2017) Anderson-darling test. [Online]. Available: https://en.wikipedia.org/wiki/Anderson%E2%80%93Darling_test

Acknowledgments

I would like to thank my supervisor, without whom I wouldn't have had the chance to work on a project like this.

I would like to thank my relatives and my family: my mother, my father and my sister for everything they've done for me in these years, for their support and because I know they've always given me the best they could.

I also would like to thank all my dearest friends, that have been around me during good and bad times and have always supported me (even though sometimes I know it was not easy) during my journey up to here.

Lastly I would like to thank two people in particular. One of them has been with me for a long time, we've been through a lot together, he is almost like a brother for me and I'll always carry him in my heart. The other has been with me only for five years, but during those few years I've grown in a way I didn't think was possible. Thanks to her I finally managed to fully be myself and I'll never be grateful enough.

I couldn't be happier for who and where I am now and it's all been thank to you guys.