A STRATEGY FOR EVALUATING THE QUALITY OF TRACE ALIGNMENT TOOLS BASED ON A MARKOV MODEL

by

XIAO BO

A thesis submitted to the

Graduate School-New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical & Computer Engineering

written under the direction of

Professor Ivan Marsic

and approved by

New Brunswick, New Jersey,

October 2014

ABSTRACT OF THE THESIS

A STRATEGY FOR EVALUATING THE QUALITY OF TRACE ALIGNMENT TOOLS BASED ON A MARKOV MODEL

BY XIAO BO

Thesis Director:

Professor Ivan Marsic

Trace alignment of event logs is used to understand and improve business processes. A key missing component of current approaches for performing trace alignment is a methodology to measure the quality of alignment. We propose a novel approach for generating random event logs that can be used for testing and evaluating trace alignment tools. We first extracted a statistical model from 437 real-world traces from Children’s National Medical Center in Washington DC (CNMC). We then created a guide tree and prune it to a minimum spanning tree based on user defined trace number in output event log. The final step is to fill this tree. Each node in this tree contains a trace. Each leaf node represents a trace in output event log. The root node is filled by a user-defined sequence and each child node is mutate from parent node based on the statistical model. To validate our approach, we used a concept of replay fitness score. Replay fitness score is used to quantify the extent to which a model can reproduce the traces recorded in an event log. It’s between 0 and 1. The value 1 means that the model can perfectly replay the event log and 0 means that the model cannot reply the log. Comparing with process model (Petri-Net) extracted from 437 real-world traces, the output event log of our system can constantly get a score of 0.8.

Therefore, our results are relevant not for only validation of trace alignment tools but also for other process mining tools.

ACKNOWLEDGEMENT AND DEDICATION

I would like to thanks Professor Ivan Marsic for his wonderful mentoring. His influence can be seen throughout the thesis. I’m also thankful to Dr. Jaroslaw Zola, Dr.Burd and Dr. Jens Stoye. Their insightful input also inspired my new idea. I’m also grateful for my parents for their encouragement. Without their support, I can hardly reach the completion of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 : INTRODUCTION

1.1 Background and Motivation

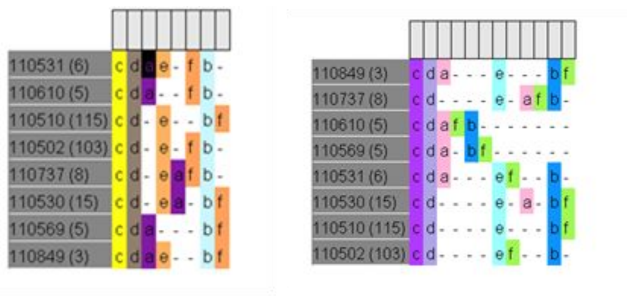
In many processes, events can be recorded in an event log in which the events within a particular process execution (“trace”) and their time of execution (“timestamp”) are recorded. The event log can then be used to determine the pattern of sequences in which a process is executed. Determining these sequences is particularly challenging for complex processes for which the steps of the process are not well-known or when real-world uncertainties introduce a large amount of variability in process execution. Mining complex event logs for execution patterns and non-standard activities, however, may be important for understanding workflow and improving process quality and productivity.

Process mining approaches have been developed for identifying successful and unsuccessful patterns of process execution and identifying deviations within these processes that may be associated with adverse events [19]. A recent process mining approach for performing this type of analysis is “trace alignment,” where the traces of events are aligned to identify the flow of events corresponding to the standard practice (known as “consensus sequence”) [3]. Because patient-evaluation tasks during trauma resuscitation are standardized and prioritized, it appropriate to apply trace alignment on resuscitation event logs. Medical experts could use trace alignment techniques to visualize the event traces and discover the consensus sequence to understand the “average” case. Trace alignment is widely used for studying genetic sequences and there are well-established techniques for evaluating the quality of trace alignment algorithms [22]. However, trace alignment of business process logs is new and such techniques do not yet exist. This thesis focuses on developing a quality measure for trace alignment algorithms.

Our problem domain is trauma resuscitation — the initial care of severely injured patients in the emergency department. In trauma resuscitation, the ordering of tasks for evaluating the patient is essential to appropriate identification and management of potentially life-threatening injuries. It has been observed that critically-injured patients have up to a four-fold higher risk of death from errors than general hospital patients [17], with nearly half of these preventable deaths related to errors that occur during the initial resuscitation phase of treatment [7][12]. To standardize the resuscitation process and reduce variability, trauma centers around the world have adopted a protocol (the Advanced Trauma Life Support protocol [ATLS]), which prioritizes evaluation and management tasks [9]. The key components of the first phase of ATLS (the primary survey) can be followed using the acronym “ABCDE”, represent the steps of airway evaluation (airway or “A”), evaluating respiratory mechanics (breathing or “B”), assessing and managing circulatory status (circulation of “C”), evaluating neurological status (disability of “D”) and ensuring the patient is properly exposed (exposure or “E”). Given the potential benefits of ensuring ATLS protocol compliance, our long-term goal is to build an automatic decision-support system that will help reduce human errors during trauma resuscitation.

1.2 Problem Statement

Trace alignment strategies have been developed for business process mining but these strategies can yield diverse results. No established methods no exist for evaluating the quality of these alignment results. As an example, an alignment of 268 ATLS traces was performed for a set of ATLS tasks using two different trace alignment strategies using a plugin in ProM 6.3[1] (Figure 1). Despite using the same initial data set, different results were obtained when a block-shift alignment algorithm (right of Figure 1) or regular alignment algorithm [3] that does not use block-shift alignment (left of Figure 1) was used. For example, in these two alignments the trace numbered “110531” has different length and position of gaps (“-”). These differences will also make their consensus sequence different as well. These two alignments are very different and their consensus sequence are different as well.



```
110531 (6)    c d a e - f b -
110610 (5)    c d a - - f b -
110510 (115)  c d - e - - b f
110502 (103)  c d - e - f b -
110737 (8)    c d - e a f b -
110530 (15)   c d - e a - b f
110569 (5)    c d a - - - b f
110849 (3)    c d a e - - b f
```

```
110849 (3)    c d a - - - e - - - b f
110737 (8)    c d - - - - e - a f b -
110610 (5)    c d a f b - - - - - - -
110569 (5)    c d a - b f - - - - - -
110531 (6)    c d a - - - e f - - b -
110530 (15)   c d - - - - e - a - b f
110510 (115)  c d - - - - e - - - b f
110502 (103)  c d - - - - e f - - b -
```

Figure 1: Example trace alignments for the same event log using two different algorithms available in ProM 6.3. The left chart was produced using a regular alignment algorithm and the right chart additionally used the block shift algorithm. The letters stand for: c = airway assessment, d = breath sounds, a = palpation of central pulses, e = palpation of distal pulses, b = Glasgow Coma Score (GCS) assessment, f = pupil exam.

[1] ProM 6.3 website: http://www.promtools.org/prom6/

Given the results of trace alignment may differ based on the method used, it is necessary to have a benchmark that can measure the quality of output of each trace alignment algorithm. We propose that this type of a benchmark should address the following two issues:

● Generating Artificial Event Logs: The tool should be able to generate random event logs based on the process model. During a business process execution (in our case, trauma resuscitation), the workers may not follow the standard ordering of tasks because of errors or other reasons. The tool should be able to reproduce the kind of variability present in real executions of the process.

● Consensus Sequence Evaluation: The tool should be able to evaluate the quality of the computed consensus sequence by comparing it with each trace in the input event log.

1.3 Related Work

In the area of process mining, the main approach to creating benchmark datasets is based on a generative framework for representing well-understood processes (e.g., PLG Framework [5]). The approach is based on a context-free grammar of commonly observed patterns of event sequences, such as “sequence,” “split and merge,” or “loop”. Each production of this grammar is associated with a probability of occurrence of a pattern in the current step. This approach may be suitable for describing well-understood existing processes or newly designed processes. However, it may not be suitable for complex and poorly understood processes such as trauma resuscitation. Because process patterns and corresponding probability distributions are unknown, the generated event log may not be a realistic representation of actual process executions.

In a related area, previous work has been developed for generating test datasets for DNA, RNA, and protein sequence alignment (e.g., ROSE [18]). This approach is based on a phylogenetic evolution tree and a substitution model (e.g., HKY85[25]) derived from known statistics of evolution process at the molecular sequence level. It also considered insert-delete (INDEL) operations and “sequence motifs” [26] which represent regions of functional importance, where mutation is less likely to happen. The knowledge of these facts is critical to simulating the evolution process. A limitation of this approach is that it mainly focuses on evolution of each position in a given sequence but overlooks the dependencies between adjacent positions. Each position is mutated independently of the adjacent positions, reflecting the nature of genetic mutations. On the other hand, in business process the next task often depends on the current task. Therefore, we need to consider the task in the current position when selecting the “mutation” for the next position (the next task).

Sequence motif is another major factor related to the first issue. In the area of computational biology, MEME (Multiple EM for Motif Elicitation) is a widely used tool for searching repeated and un-gapped sequence patterns in a set of DNA/Protein sequences[27]. It chooses each motifs by minimizing the probability of finding an equally well-conserved pattern in random sequences. This approach may be suitable for well-understood DNA/Protein sequences. But it may not be suitable for poorly-understood and complex process traces such as ATLS traces.

Related to the second issue of consensus sequence evaluation, two main groups of string or sequence metrics can be used [16]. The first group includes the Hamming distance metric[24], which is computationally very fast and has a low memory requirement, but they may not reflect the actual similarity between two sequences. The second group includes the edit distance (based on Levenshtein distance), which can precisely measure the similarity between two sequences but these metrics are less efficient in terms of processing speed and memory requirements.

This thesis addresses the two key issues identified earlier. First, we propose a novel methodology to generate artificial event logs of the trauma resuscitation process that can serve as test event logs for trace alignment tools. Our method is based on an equivalent of evolution tree in genetics. We can think of the “ideal” execution of the resuscitation process that is prescribed by ATLS as equivalent of the “ancestor sequence”. Any variations observed in actual resuscitations can be thought of as “mutations of the ancestor sequence” The statistical parameters of our method are derived from real resuscitations logs. Second, we propose a methodology to measure the quality of consensus sequence based on Phred-quality score [10] and edit distance[23].

CHAPTER 2 : PROCESS MODEL AND DATA SOURCE

2.1 Data Source

Our input event log is generated from video recordings at the trauma center of the Children’s National Medical Center (CNMC) in Washington, DC. The data were collected over from events that occurred over eight months. The resulting log contains ATLS event traces from 437 trauma resuscitations as well as a timestamp allowing assessment of the order of event performance [8]. The approach that was used for obtaining this event log has been previously described [13][14].

2.2 Declarative Model and Statistical Model

A declarative model of the process for which the traces were collected was developed by medical experts based on the knowledge of the ATLS process [2] performed by the bedside evaluator (Figure 2). It included six primary ATLS survey tasks. This model represents the best practice for ordering and inclusion of these tasks. For example, “breath sounds” should not happen before “airway assessment” is accomplished and “GCS assessment” should not occur unless either the “central pulse” or “distal pulses” are measured. Although this is a best-practice model, actual process execution may not follow this model. First, the provider may go back to previous steps to reevaluate an observed change. Second, the process may be performed out of order because of situational variables such as the availability of needed evaluation tools or personnel.

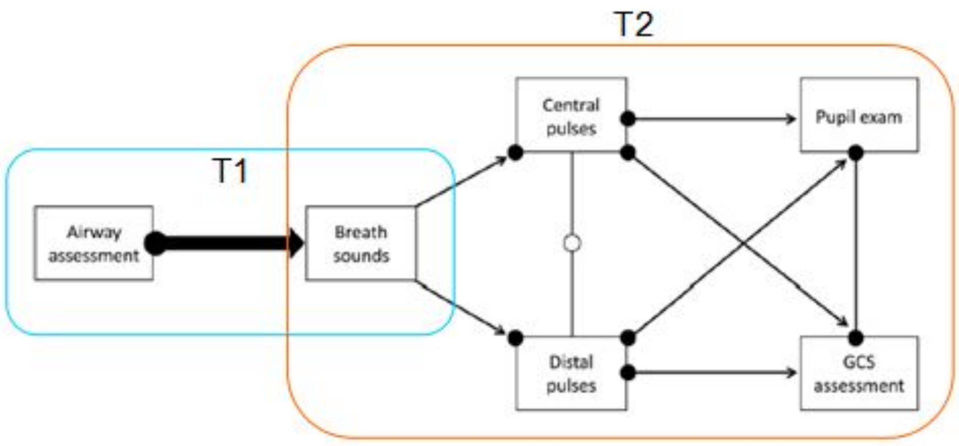


Figure 2: Declarative model of six key tasks of the ATLS primary survey.

We can state the above description in a more formal way by representing a real-world business process using a first-order Markov model. Given a trace with M events, the ith event (1(i(M) depends on the (i(1)st event, for i>1. The parameters of the proposed model are as follows:

● Event frequencies PREFIX_FREQ(f1, f2, ..., fn) satisfying $\sum_{i=1}^{n} f_i = 1$ are used for creating the first event in a trace.

● Mutation matrix T1 represents the transition probability from the first to the second event.

● Mutation matrix T2 represents the transition probability from event i to event i(1 (i(2).

● Length frequencies L(ℓE1, ℓE2 , ..., ℓEn), represent the possible lengths of traces that start with different events E1, E2, ..., En. Given a first event of type Ei in a trace, different possible lengths k of the trace occur with different frequencies fk. Therefore, ℓEi(f1, f2, ..., fn) represents the probability distribution of different trace lengths, given the first event = Ei and the distribution satisfies $\sum_{k=1}^{n} f_k = 1$. Let us assume that an event trace could start with these first events: E1= “airway assessment” or E2= “ breath sounds. ” Then we need to specify L = { ℓ airway-assessment, ℓ breath-bounds}. For example, traces that start with “airway assessment” and be up to five events long, but only trace lengths 2 and 5 occur, and they occur equally likely. Then, the probability distribution for ℓ airway-assessment = {f1 = 0, f2 = 0.5, f3 = 0, f4 = 0, f5 = 0.5}.

The model parameters (PREFIX_FREQ, T1, T2, L) are extracted from a given event log as follows:

```
**Function** get_prefix_freq(log) **return** PREFIX_FREQ:
PREFIX_PREQ = {}        // PREFIX_FREQ is a Python-like
                        // dictionary. It could also be implemented by a hash-table.
Counter = 0
For trace in log:
    First_event = trace[0]
    If first_event not in PREFIX_PREQ:
        PREFIX_FREQ[first_event] = 0
    End if
    PREFIX_FREQ[first_event]++
    Counter++
End for
For event in PREFIX_FREQ:
        PREFIX_FREQ[event] = PREFIX_FREQ[event] / Counter
End for
Return PREFIX_FREQ
```

Other parameters of the model are extracted similarly. Table 1 and Table 2 show the extracted contents of T1 and T2. The letter encoding is as defined in Figure 1.

Next Previous	c	d	a	e	b	f
c	0	0.01	0.06	0.52	0.16	0.16
d	0.003	0	0.09	0.79	0.07	0.05
a	0.02	0.04	0	0.18	0.41	0.43
e	0.04	0.002	0.1	0	0.41	0.43
b	0.004	0.03	0.05	0.07	0	0.85
f	0.02	0	0.07	0.06	0.85	0

Table 2: Contents of T2, where the letter codes are as defined in Figure 1. T2[c,d] means the probability of i_{th} event = "d=breath sounds" if $(i-1)_{th}$ event "c=airway assessment," where 1<i<trace-length.

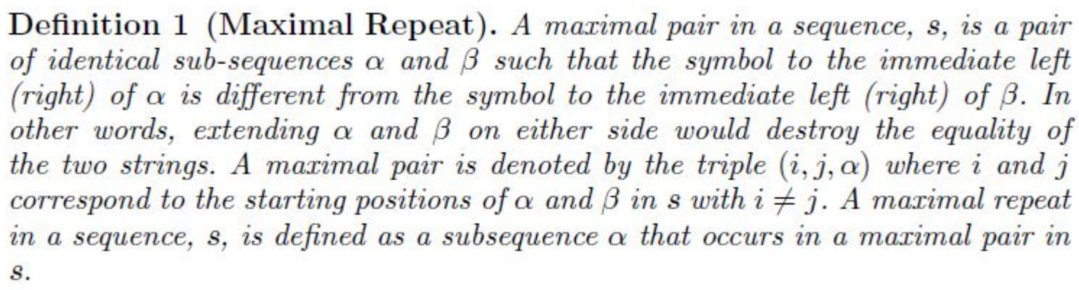
NEXT PREVIOUS	c	d	a	e	b	f
c	0	0.95	0.01	0.02	0.02	0
d	0.6	0	0.03	0.3	0.03	0.03
a	0	0	0	0	0	0
e	0	0	0	0	0	0
b	0.7	0.3	0	0	0	0
f	1	0	0	0	0	0

Table 1: Contents of T1, where the letter codes are as defined in Figure 1. T1[c,d] means the probability of position 1 being “d=breath sounds” if position 0 was “c=airway assessment.”

The motivation to partition the mutation matrix into T1 and T2 is to avoid the possibility that noise present in T2 affects T1. For example, based on the declarative model (Figure 2), “pupil exam” and “GCS assessment” are not acceptable in the first two steps, but are highly coupled. Based on the event log of 437 observed traces, their mutual transition probabilities after the first two steps (“pupil exam (GCS assessment” and “GCS assessment (pupil exam”) are (0.85. However, some real traces started with “pupil exam” (not adhering to the model in Figure 2), which was invariably followed by “airway assessment” (see that T1[f, c] = 1 in Table 1). In such cases, if we combined all transition probabilities into a single mutation matrix, the system could generate a trace like “pupil exam (GCS assessment (pupil exam (GCS assessment”, which is unacceptable because no such traces were observed. In other words, statistics from the “normal” cases (represented by the model) could distort the transition probabilities.

2.3 Sequence Motif Mining

In the area of process mining, sequence motif is considered as common patterns of invocations of activities in traces[28]. It's key to reflect the real execution of process model. Currently, several patterns are defined in [28]. In this paper, we utilize maximal repeats as an example. The followings are its definition[28][29]:



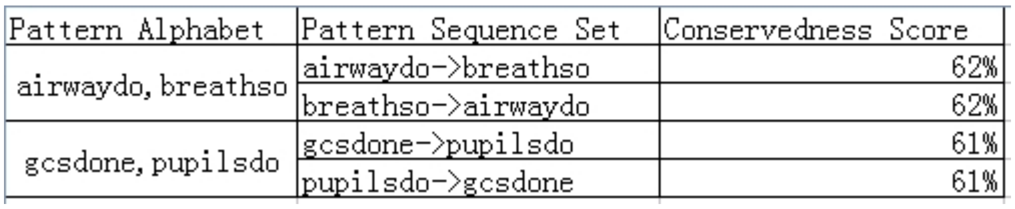
Definition 1 (Maximal Repeat). *A maximal pair in a sequence, s, is a pair of identical sub-sequences α and β such that the symbol to the immediate left (right) of α is different from the symbol to the immediate left (right) of β. In other words, extending α and β on either side would destroy the equality of the two strings. A maximal pair is denoted by the triple (i, j, α) where i and j correspond to the starting positions of α and β in s with $i \neq j$. A maximal repeat in a sequence, s, is defined as a subsequence α that occurs in a maximal pair in s.*

It captures the common sequence of activities among traces and might be evidence of common functionality[29]. It can be efficiently mined by suffix tree within linear time[30][31].

We also introduced the concept of Conservedness to measure the confidentiality of each motif. It measures the degree to which the individual activities involved in the pattern alphabet manifest as the patterns defined by the alphabet[29]. The followings are its definition[29]:

$\text{Conservedness} = \frac{\text{NOAC}}{\mu} * (1 - \frac{\alpha}{\mu}) * 100\%$, where NOAC is Non-Overlapping Alphabet Count, μ and α are the mean and standard deviation of frequencies of activities.

In actuality, we used "Pattern Abstraction" plugin to calculate sequence motifs set M from 437 actual traces. When calculating NOAC, we prefer shorter patterns. All duplicate traces are also ignored. All patterns whose length == 1 or conserved-ness score is less than 50% are also pruned. Table 3 are discovered sequence motifs and their conserved-ness score:



Pattern Alphabet	Pattern Sequence Set	Conservedness Score
airwaydo, breathso	airwaydo->breathso	62%
	breathso->airwaydo	62%
gcsdone, pupilsdo	gcsdone->pupilsdo	61%
	pupilsdo->gcsdone	61%

Table 3: Contents of M, where "Pattern Sequence Set" Column represents each discovered sequence motifs.

CHAPTER 3 : ALGORITHM

3.1 Overview

Our system consists of two components:

1. Trace Generator creates a random event log used in trace alignment
2. Consensus Evaluator evaluates the generated consensus sequence.

3.2 Trace Generator

The trace generator produces a random event log simulating real-world variability of the business process. It is common to represent this process by a guide tree, where each node contains a trace (Figure 3). The root node (Node 0) contains an “ideal” trace of standard ordering of tasks, and the other nodes contain variations. When executing the process in a standard way, errors may occur in the actual execution would “mutate” from Node 0 to Node 1. For instance, suppose the trace in Node 0 is “airway assessment -> breath sounds -> central pulse -> GCS assessment”. “Breath sounds” may have been skipped in the trace in Node 1 and Node 1 become “airway assessment -> central pulse -> GCS assessment” . As more mistakes occur, the actual execution may further “mutate” from Node 1 to Node 4.

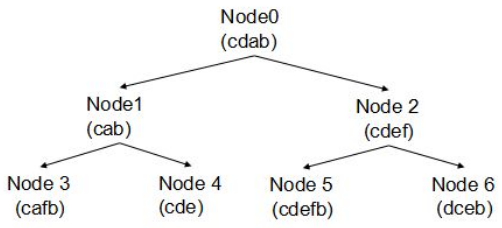


Figure 3: An example guide tree used for simulating real-world variability of trauma resuscitation. The letter encoding for example traces in the parentheses is as defined in Figure 1.

In addition to (PREFIX_FREQ, T1, T2, L, M), the generator requires the following input:

● An Ancestor Trace S: This trace is the standard trace used to generate variations. This trace should be user-defined. If the user does provide the standard trace (e.g., because of a poorly specified process), a random trace of length ℓ will be generated based on the model parameters (PREFIX_FREQ, T1, T2 and L).

● Edge Distance D: This number represents the frequencies of mutation from parent node to its children nodes. In Figure 3, if D = 8 Node 1 mutates eight times before reaching Node 4. The edge distance is used to create a mutation guide tree which acts as a mapping for generating a random event log [18].

● Revival Probability R: This number represents the probability to generate a child trace without considering the parent trace. For instance, Node 4 may be created as a random trace rather than a mutation of Node 1. The main motivation for this feature is to increase the randomness of output datasets.

- Trace Number N: This number determines the number of traces in the output event log E (defined below).
- Tree Depth d: This number represents the depth of the guide tree, i.e., how many generations the tree will evolve away from the root trace. For example, in Figure 3, d = 2.

The output of the trace generator is:

- Event Log E: The event log contains all random traces that are generated by the trace generator.

The following is the pseudocode of the algorithm:

```
Function trace_generator(S, D, d, N, R, PREFIX_FREQ, T1, T2, L, M) return E
If S is undefined
    S = create_sub_trace(null, 0, n, PREFIX_FREQ, T1, T2, L)
End if

T = create_guide_tree(D, d)
prune_tree(T, N)

T.root = S
```

```
fill_tree(T.root, M)
```

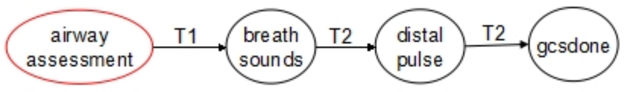


Figure 4: Generation of a sample sub trace.
In this case, seed event *E* = airway assessment, start position *P* = 0, and trace length ℓ = 4.

```
Traces = get_traces(T)

Return Traces
```

Function fill_tree() is implemented as a recursion:

```
Function fill_tree(T.root, M)

For children of T.root do

    Children.seq = mutate(T.root)

    Define_motif_positions(Children, M)

fill_tree(Children)

End for
```

The remaining functions are defined in the following subsections.

3.2.1 Generating the Sub Traces

The function create_sub_trace(E, P, ℓ, PREFIX_FREQ, T1, T2, L) is the core function for generating event logs. It returns a trace of length ℓ. In addition to (PREFIX_FREQ, T1, T2, L), it also requires the following inputs:

- Seed Event E: The first event in the trace or sub-trace that will be generated.

● Start Position P: The position of the seed event in the original trace. This number will determines what the transition model (T1, T2) will be used for generating a sub-trace.

● Trace Length ℓ: The length of the target (output) sub-trace.

Figure 4 shows an example creation of a sub-trace. Because the seed event is also the first event in the parent-node trace, the second event in a sub-trace is determined by E and T1 based on the model (Figure 2). The third event is based on T2 and the second event, and so is the fourth event. If the input seed event is null, the seed event will be generated from PREFIX_FREQ.

3.2.2 Generating the Mutation Guide Tree

Creation of the mutation guide tree is done in the function create_guide_tree(). After the tree is created, N nodes are randomly marked as “selected,” because the user requested N traces in the output event log E. To reduce the computational complexity, we compute the minimum spanning tree and prune the tree. For example, in Figure 3, when Nodes 3 and 4 are selected, Nodes 2, 5, and 6 will be removed because they are not relevant to reaching Nodes 3 and 4 from the root node. After the minimum spanning tree is generated, the system generates the output event log E from the root node by fill_tree() and get_traces().

3.2.3 Generating Children Traces

The core function for children trace creation is the function mutate(). The followings are its main steps:

1. Generate trace of length ℓ based on the type Ei of the first event in the trace: ℓ = getLength(Ei, L)

2. Determine the mutation start position STARTPOS ($0 \leq STARTPOS < \ell$)

3. Create sub-trace based on T_{old}[STARTPOS]:

$$T_{sub} = \text{create\_sub\_trace}(T_{old}[\text{STARTPOS}], \text{STARTPOS}, \ell - \text{STARTPOS}, \text{PREFIX\_FREQ}, \text{T1}, \text{T2}, \text{L})$$

4. Create child trace by removing the old sub-trace starting at position STARTPOS and appending instead it the created sub-trace:

$$T_{old}.\text{RemoveAll}(\text{STARTPOS}, \text{Told.length});$$

$$T_{new} = \text{Told.append}(T_{sub})$$

By above steps, the INDEL operations are realized since L contains all possible trace lengths. The sequence motif information is also represented in T1, T2. These steps are repeated for edge distance D times in order to maintain the randomness of child traces.

3.2.4 Creation of Sequence Motif

Creation of sequence motif is implemented by define_motif_positions(). When defining sequence motifs in children trace, we search each pattern in M and mark corresponding positions as “important”. These “important” positions will determines the penalty score in consensus sequence evaluation.

3.3 Consensus Sequence Evaluator

The quality of consensus sequence is evaluated from two aspects: confidence about the element in each position and similarity between traces in event log and consensus sequence. Phred-quality score is used to measure the confidence for each position:

$Q = -10*\log10(P)$, where Q represents Phread-quality score and P represents base-calling error probability [10][11].

Q, however, only estimates the confidence of the element in each position in the consensus sequence, independently of other positions. For this reason, we use regular edit distance to measure similarity between the consensus sequence and individual traces in test event logs instead of Q. “Sequence motifs” which represent critical positions in the root trace that are subject to lower mutation rates compared to other positions. Therefore, we add a “penalty” in the quality score formula, to penalize mutations in the corresponding positions. The followings are formula to compare consensus sequence and each trace:

$Q_2 = \text{Edit\_Distance} + X * N_{dif}$, where X is a user-defined coefficient and N_{dif} represents number of mutated critical positions contains in consensus sequence.

CHAPTER 4 : RESULTS AND DISCUSSION

To test our trace generator, we used the concept of Replay Fitness Score [4][19]. This score is used to quantify the extent to which a model can reproduce the traces recorded in an event log. The score is computed as [4]:

$$Q_{rf} = 1 - \frac{\text{cost for aligning model and event log}}{\text{Minimal cost to align arbitrary event log on model and vice versa}}$$

where Qrf is between 0 and 1. The value 1 means that the model can perfectly replay the event log and 0 means that the model cannot reply the log.

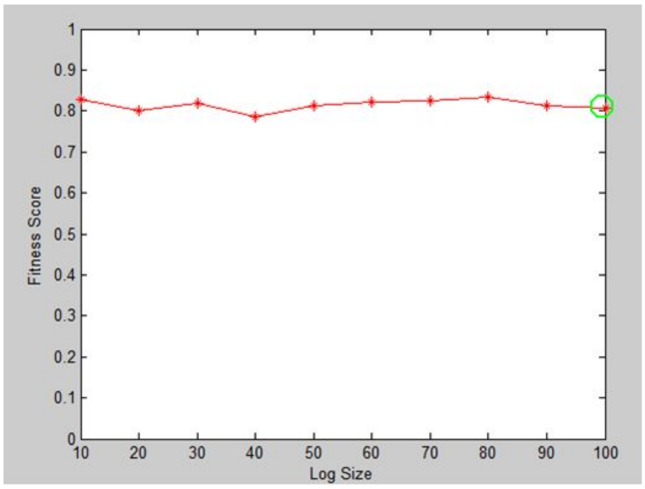


Figure 5: Replay Fitness Score of artificial event logs generated by our tool, compared to the Petri-net model created by ProM from 437 actual event traces.

To explore the properties of our trace generator, we generated 10 event logs with 10 to 100 traces each. The ancestor trace S is “airway assessment -> breath sounds -> pupil exam -> GCS assessment -> pupil exam -> GCS assessment” and the guide tree is created with the parameters d = 10, D = 1 and R = 0.5 (Section 3.2). We also computed an approximate model (Petri Net) of 437 traces by using the ProM plugin named “Mine a Petri net using ILP” [21]. Next, we computed the fitness score using the ProM plugin named “Replay a log on Petri Net for conformance checking.” The fitness score determines how well our artificially generated event log fits the Petri Net model of the actual event log [1][20]. The core algorithm is a cost-based A* algorithm [1]. The fitness score Qrf for our event log generator is around 0.8 (Figure 5). Based on the definition of Qrf, it should be equal to 1 for all artificial event logs that perfectly simulate the real-world process execution. A potential reason that event logs generated by our tool generally have lower fitness score (Figure 5) is that the Integer Linear Programming (ILP) approach performs well in terms of speed but lacks in precision [21].

To further understand the characteristics of our trace generator, we used one of event logs from Figure 5. This event log had 100 artificial traces and its fitness score is circled by green in Figure 5. We computed its consensus sequence C1 using the trace alignment plugin of ProM as C1 = “airway assessment -> breath sounds -> GCS assessment -> pupil exam.” The plugin uses the Euclidean distance [8] as part of the alignment process. We also computed the consensus sequence C2 of the 437 actual traces using same algorithm and obtained C2 = “airway assessment -> breath sounds -> distal pulse -> GCS assessment -> Pupil Exam.” When these two consensus sequences are compared, the only difference is that in C2 “distal pulse” is missing (Figure 6). Based on T2 (Table 2), both “GCS assessment” and “distal pulse” can follow after a “breath sounds” event, but the probability T2[breath sounds, distal pulse] is much greater than T2[breath sounds, GCS assessment].

Therefore, the consensus sequence C1 of our artificial event log should not have omitted " distal pulse " if our tool perfectly simulated the real-world process execution. There are two potential explanations for this shortcoming. First, the trace alignment algorithm in ProM [3] may not be robust enough to handle some real-world variability of event traces. Second, 100 traces in our artificial event log or 437 actual traces may be insufficient for accurate representation of statistical properties of the trauma resuscitation process.

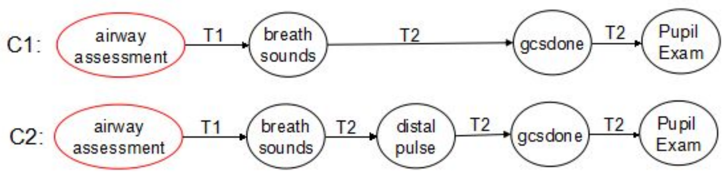


Figure 6: Comparison of the consensus sequence for an artificially-generated event log (C1) to that of the 437 actual event traces of trauma resuscitation (C2).

We used Phred-quality score for consensus sequence evaluation, which has proved to provide a robust metric for evaluating the performance of alignment tools for genetic sequences [10][11]. Phred-quality score to measure the confidence about each position in a consensus sequence. However, since our problem domain is different, this issue needs further exploration.

CHAPTER 5 : CONCLUSION AND FUTURE WORK

The event logs created by our methodology are artificial datasets used for evaluating how the ProM trace alignment tool performs on a given event log. Our experimental results showed that our tool can generate random event logs that can replay the real execution of a declarative model of ATLS. Our tool combines the strengths of several existing evaluation frameworks (e.g., PLG and ROSE), such as using guide tree and process model to generate realistic event logs of the trauma resuscitation process. Unlike PLG, we use the statistical model extracted from real data and a guide tree to event traces. Unlike Rose, which mutates each position in the sequence independently of its neighbors, we consider the current task when generating the next task. In this sense, our tool “grows” the event traces from starting event to the final event. Furthermore, we extracted sequence motifs from real data and defines the positions of functional importance. For these reasons, our tool is suitable for evaluating trace alignment tools for their potential use for studying event logs of trauma resuscitations.

Our model was designed to simulate the event traces resulting from real execution of an ATLS-based process. We plan to improve this evaluation tool in several ways. First, the model for INDEL operation is unrealistically simple. Currently, we randomly choose a position in the trace and then grow a new sub-trace from it. However, not all positions have the same probability that a mutation will occur. For instance, about 78% of our dataset of 437 traces started with “airway assessment”, and about 95% of those traces which started with “airway assessment” had “breath sounds” in the second position. Second, as discussed in Section 4, we need to prove whether Phred-Quality Score is accurate for consensus sequence evaluation. Third, some combinations of tasks are critical and are less likely to vary (i.e., mutate), because the workers are trained to pay special attention to these tasks. However, statistical commonly sequence pattern may not contains all of these information. To address this potential finding, we also plan to introduce domain-knowledge and define “motifs,” and suppress mutation on these tasks.

REFERENCE

[1] Adriansyah, A., van Dongen, B.F., and van der Aalst, W.M.P., “Conformance checking using cost-based fitness analysis,” ACSD 2011: 57-66.

[2] American College of Surgeons Committee on Trauma and Institutional Practice, ATLS: Advanced trauma life support for doctors (student course manual), 9th ed., American College of Surgeons, Chicago, IL, 2012.

[3] Bose, R.P.J.C., van der Aalst, W.M.P., “Process diagnostics using trace alignment: opportunities, issues, and challenges,” Inform Syst 2012;37:117-141.

[4] Buijs, J.C.A.M., van Dongen, B.F., and van der Aalst, W.M.P., “On the role of fitness, precision, generalization and simplicity in process discovery,” On the Move to Meaningful Internet Systems: OTM 2012.

[5] Burattin, A., and Sperduti, A., “PLG: A framework for the generation of business process models and their execution logs,” BPM 2010 Workshops, LNBIP 66, pp. 214-219, 2011.

[6] Carter, E.A., Waterhouse, L.J., Kovler, M.L., et al. “Adherence to ATLS primary and secondary surveys during pediatric trauma resuscitation,” Resuscitation 2013;84:66-71.

[7] Demetriades D, Kimbrell B, Salim A, Velmahos G, Rhee P, Preston C, Gruzinski G, and Chan L., “Trauma deaths in a mature urban trauma system: Is ‘trimodal’ distribution a valid concept?” J Am Coll Surg. 2005 Sep;201(3):343-348. PubMed PMID: 16125066.

[8] Deza, E., and Deza, M.M., Encyclopedia of Distances, p. 94, Springer-Verlag, 2009.

[9] Driscoll, P. and Wardrope, J: “ATLS: past, present, and future,” Emerg Med J 2005;22:2-3 doi:10.1136/emj.2004.021212

[10] Ewing B., Green, P. “Base-calling of automated sequencer traces using Phred. II. Error probabilities,” Genome Res. 8 (3): 186-194, 1998.

[11] Ewing, B., Hillier, L., Wendl, M.C., and Green, P., “Base-calling of automated sequencer traces using Phred. I. Accuracy assessment". Genome Res. 8 (3): 175-185, 1998.

[12] Gruen, R.L., Jurkovich, G.J., McIntyre, L.K., Foy, H.M., and Maier, R.V., “Patterns of errors contributing to trauma mortality: lessons learned from 2,594 deaths,” Ann Surg. 2006 Sep;244(3):371-80. PubMed PMID: 16926563; PubMed Central PMCID: PMC1856538.

[13] Kelleher, D.C., Bose, R.P.J.C., Waterhouse, L.J., Carter, E.A., and Burd, R.S., “Effect of a checklist on advanced trauma life support workflow deviations during trauma resuscitations without pre-arrival notification,” Journal of the American College of Surgeons, 2014 Mar;218(3):459-66. doi: 10.1016/j.jamcollsurg.2013.11.021. Epub 2013 Nov 26.

[14] Kelleher, D.C., Carter, E.A., Waterhouse, L.J., and Burd, R.S., “Compliance with barrier precautions during pediatric trauma resuscitations,” Resuscitation, vol. 84, pp.314-318, 2013.

[15] Kelleher, D.C., Kovler, M.L., Waterhouse, L.J., et al. “Factors affecting team size and task performance in pediatric trauma resuscitation,” Pediatr Emerg Care, In press.

[16] Seker, S.E., Altun, O., Ayan, U., and Mert, C., “A novel string distance function based on most frequent k characters”, International Journal of Machine Learning and Computation (IJMLC), vol. 4, no. 2, pp.177-183, 2014.

[17] Stahl, K.D., and Brien, S.E., “Reducing patient errors in trauma care,” In: Cohn SM, ed. Acute Care Surgery and Trauma Care. London, UK: Informa Health Care, 268-277, 2009.

[18] Stoye, J., Evers, D., and Meyer, F., “Rose: generating sequence families,” Bioinformatics, vol. 14, no. 2, pp. 157-163, 1998.

[19] van der Aalst, W.M.P. Process Mining: Discovery, Conformance and Enhancement of Business Processes, Springer-Verlag Berlin Heidelberg, 2011.

[20] van der Aalst, W.M. P., Adriansyah, A., van Dongen, B.F., “Replaying history on process models for conformance checking and performance analysis,” Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery 2(2): 182-192, 2012.

[21] van der Werf, J.M.E.M., van Dongen, B.F., van Hee, K.M., Hurkens, C.A.J., and Serebrenik, A. “Process discovery using integer linear programming,” Applications and Theory of Petri Nets, vol. 5062, pp 368-387, 2008.

[22] Waterman, M.S., Introduction to Computational Biology: Maps, Sequences and Genomes, Chapman & Hall, CRC, 2000.

[23] Eric Sven Ristad,Peter N. Yianilos, “Learning String-Edit Distance”, IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, VOL. 20, NO. 5, MAY 1998

[24] R.W. Hamming, “Error Detecting and Error Correcting Codes”, The Bell System Technical Journal, Vol, XXVI, April 1950

[25] Hasegawa M, Kishino H, Yano T . "Dating of human-ape splitting by a molecular clock of mitochondrial DNA". Journal of Molecular Evolution22 (2): 160–174,1985

[26] Matsuda H, Taniguchi F, Hashimoto A. "An approach to detection of protein structural motifs using an encoding scheme of backbone conformations". Proc. of 2nd Pacific Symposium on Biocomputing: 280–291, 1997

[27] Timothy L. Bailey, Nadya Williams1, Chris Misleh1 and Wilfred W. Li: “MEME: discovering and analyzing DNA and protein sequence motifs”, Nucleic Acids Research, 2006, Vol. 34, Web Server issue W369–W373

[28] Jagadeesh Chandra Bose, R.P. & Aalst, W.M.P. van der (2009):“Abstractions in process mining : a taxonomy of patterns”, In U. Dayal, J. Eder, J. Koehler & H.A. Reijers (Eds.), Business Process Management (7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009. Proceedings) (pp. 159-175). Berlin: Springer-Verlag.

[29] Jiafei Li, R. P. Jagadeesh Chandra Bose, Wil M. P. van der Aalst: “Mining Context-Dependent and Interactive Business Process Maps using Execution Patterns”, BPM 2010 International Workshops and Education Track, Hoboken, NJ, USA, September 13-15, 2010, Revised Selected Papers, pp 109-121

[30] Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997)
[31] Kolpakov, Kucherov: Finding Maximal Repetitions in a Word in Linear Time. In: IEEE Symposium on Foundations of Computer Science (FOCS). (1999) 596-604

APPENDIX A : ALGORITHM TO EXTRACT STATISTICAL MODEL

1. Algorithm To Extract PREFIX_FREQ:

```
Function GET_PREFIX_FREQ(Event-Log) return PREFIX_FREQ:
PREFIX_FREQ = {}        // PREFIX is a dictionary
COUNTER = 0
For trace in Event-Log:
    Event = trace[0]
    If Event not in PREFIX_FREQ:
        PREFIX_FREQ[Event] = 0
    End If
    PREFIX_FREQ[Event]++
    COUNTER++
End for
For event in PREFIX_FREQ:
    PREFIX_FREQ[event] = PREFIX_FREQ[event]/COUNTER
End For
Return PREFIX_FREQ
```

2. Algorithm To Extract T1:

```
Function GET_T1(Event-Log) return T1:
T1 = {}        // T1 is a dictionary
For trace in Event-Log:
    If trace.length < 2:
        **Continue**
    First_Event = trace[0]
    Second_Event = trace[1]
    **If** First_Event not **in** T1:
        T1[First_Event] = {}
    **End if**
    **If** Second_Event **not in** T1[First_Event]:
        T1[First_Event][Second_Event] = 0
    **End if**
    T1[First_Event][Second_Event]++
**End For**
**For** first_event **in** T1:
    COUNTER = 0
    **For** second_event **in** T1[first_event]:
        COUNTER = COUNTER +    T1[first_event][second_event]
    **End for**
    **For** second_event **in** T1[first_event]:
       T1[first_event][second_event] = T1[first_event][second_event]/COUNTER
    **End for**
**Return** T1
```

3. Algorithm To Extract T2:

```
**Function** GET_T2(log) **return** T2:
```

```
T2 = {}
For trace in log:
    If trace.length <= 2:
        Continue
    End if
    For i in range(1,trace.length-1):
        Current_Event = trace[i]
        Next_Event = trace[i+1]
        If Current_Event not in T2:
            T2[Current_Event] = {}
        End if
        If Next_Event not in T2[Current_Event]:
            T2[Current_Event][Next_Event] = 0
        End if
        T2[Current_Event][Next_Event]++
    End for
For Current_Event in T2:
    COUNTER = 0
    For Next_Event in T2[Current_Event]:
        COUNTER = COUNTER + T2[Current_Event][Next_Event]
    End for
    For Next_Event in T2[Current_Event]:
            T2[Curent_Event][Next_Event] = T2[Current_Event][Next_Event]/ Counter
        End for
Return T2
```

4. Algorithm To Extract L:

```
Function GET_L(Event-Log) return PREFIX_FREQ:
L = {}        // L is a Python-like dictionary
For trace in Event-Log:
    First_Event = trace[0]
    Cur_Len = trace.length
    If First_Event not in L:
        L[First_Event] = {}
    End If
    If Cur_Len not in L[First_Event]:
        L[First_Event][Cur_Len] = 0
    End If
    L[First_Event][Cur_Len]++
End for
For event in L:
    Counter = 0
    For Cur_Len in L[event]:
        Counter = Counter + L[event][Cur_Len]
    End for
    For Cur_Len in L[event]:
        L[event][Cur_Len] = L[event][Cur_Len] / Counter
    End for
End For
Return L
```

APPENDIX B : ALGORITHM TO GENERATE CHILDREN TRACE (JAVA CODE)

```
public String[] mutate(String[] oldTrace) {

    ArrayList<String> buffer = new ArrayList<String>(

            Arrays.asList(oldTrace));

    Random dice = new Random();

    HashMap<String, Float> subDict = new HashMap<String, Float>();

    for (int i = 0; i < this.distance; i++) {

        if (dice.nextFloat() >= this.revivalProb) {

            String firstEvent = this.getEvent(this.PREFIX_FREQ);

            int length = this.getLength(firstEvent);

            buffer = this.createSubTrace(firstEvent, 0, length);

            continue;

        }

        int length = this.getLength(oldTrace[0]);

        int curLen = buffer.size();

        int startPos = 0;

        if (length >= curLen) {

            startPos = dice.nextInt(curLen);

        } else {

            startPos = dice.nextInt(length);

        }
```

```
            // System.out.println(length+" "+startPos);
            ArrayList<String> subTrace = this.createSubTrace(
                    buffer.get(startPos), startPos, length - startPos);
            buffer = new ArrayList<String>(buffer.subList(0, startPos));
            if (subTrace != null) {
                buffer.addAll(subTrace);
            }
        }
        return buffer.toArray(new String[buffer.size()]);
    }

    private int getLength(String event) {
        HashMap<String, Float> subDict = this.L.get(event);
        Random dice = new Random();
        float counter = 0, prob = dice.nextFloat();
        String length = "0";
        for (String key : subDict.keySet()) {
            counter = counter + subDict.get(key);
            if (counter >= prob) {
                length = key;
                break;
            }
    }

    return Integer.valueOf(length);

}



private ArrayList<String> createSubTrace(String seed, int startPos,

        int length) {

    HashMap<String, Float> subDict = null;

    String[] subTrace = new String[length];

    if (seed == null) {

        subDict = this.PREFIX_FREQ;

        seed = this.getEvent(subDict);

    }

    if (length == 0) {

        return null;

    }

    subTrace[0] = seed;

    // System.out.println(startPos);

    for (int i = startPos + 1; i < startPos + length; i++) {

        int realIndex = i - startPos - 1;

        String prevEvent = subTrace[realIndex];

        subDict = this.T2.get(prevEvent);

        if (i == 1) {
            subDict = this.T1.get(prevEvent);
        }
        // System.out.println(startPos+" "+(startPos+length));
        // System.out.println(prevEvent);
        String curEvent = this.getEvent(subDict);
        subTrace[realIndex + 1] = curEvent;
    }
    return new ArrayList<String>(Arrays.asList(subTrace));
}
public String getEvent(HashMap<String, Float> subDict) {
    Random dice = new Random();
    float sum = 0, prob = dice.nextFloat();
    String newEvent = null;
    for (String event : subDict.keySet()) {
        sum = sum + subDict.get(event);
        if (sum > prob) {
            newEvent = event;
            break;
        }
    }
    return newEvent;
}
```

APPENDIX C : ALGORITHM FOR COMPUTING MINIMUM SPANNING TREE (JAVA CODE)

```
public void pruneTree() {
        ArrayList<TreeNode> nodeList = this.getNodeList();
        Random dice = new Random();
        int counter = 0;
        while (counter < this.nodeNum) {
            int selectIndex = dice.nextInt(nodeList.size());
            TreeNode cur = nodeList.get(selectIndex);
            if (!cur.selected) {
                cur.selected = true;
                cur.relevant = true;
                counter++;
            }
        }
        this.root.relevant = true;
        for (int i = 0; i < nodeList.size(); i++) {
            TreeNode cur = nodeList.get(i);
            if (cur.selected) {
                while (cur != this.root) {
                    cur.father.relevant = true;
                    cur = cur.father;
                }
            }
        }
        this.compressTree(this.root);
    }

    private void compressTree(TreeNode cur) {
        if (cur == null) {
            return;
        }
        if (cur.leftChild != null) {
            if (!cur.leftChild.relevant) {
                cur.leftChild = null;
            } else {
                this.compressTree(cur.leftChild);
            }
        }
        if (cur.rightChild != null) {
            if (!cur.rightChild.relevant) {
                cur.rightChild = null;
            } else {
```

```
            this.compressTree(cur.rightChild);
        }
    }
}
```

APPENDIX D: ALGORITHM FOR FILLING THE GUIDE TREE

```
public void fillTree() {

        this.root.trace = this.ANCESTOR;

        this.defineMotifs(this.root);

        this.fillNodes(this.root);

    }



    public void defineMotifs(TreeNode t) {

        if (t == null) {

            return;

        }

        HashSet<Integer> positions = new HashSet<Integer>();

        for(String[] motif : this.MOTIFS.keySet()){

            String str = this.string2character(t.trace);

            String subStr = this.string2character(motif);

            int index = str.indexOf(subStr);

            System.out.println(index);

            while(index>=0){

                for(int i=0;i<subStr.length();i++){

                    positions.add(index+i);

                }

                index = str.indexOf(str, index + 1);
            }

        }

        t.motifs = positions.toArray(new Integer[positions.size()]);

        Arrays.sort(t.motifs);

    }



    private String string2character(String[] trace){

        StringBuilder str = new StringBuilder();

        for(String event : trace){

            str.append(event.charAt(0));

        }

        return str.toString();

    }



    private void fillNodes(TreeNode father) {

        if (father == null) {

            return;

        }

        if (father.leftChild != null) {

            father.leftChild.trace = this.mutate(father.trace);

            this.defineMotifs(father.leftChild);

            this.fillNodes(father.leftChild);
    }

    if (father.rightChild != null) {

        father.rightChild.trace = this.mutate(father.trace);

        this.defineMotifs(father.rightChild);

        this.fillNodes(father.rightChild);

    }

}
```