Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Amela Špica

Računalniško podprta glasbena spremljava

MAGISTRSKO DELO

MAGISTRSKI PROGRAM DRUGE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: dr. Andrej Brodnik

Ljubljana, 2018

University of Ljubljana
Faculty of Computer and Information Science

Amela Špica

Computer Supported Play-along Music

master's thesis

the 2nd cycle master's study programme
computer and information science

Supervisor: dr. Andrej Brodnik

Ljubljana, 2018

Copyright. The results of this master's thesis are the intellectual property of the author and the Faculty of Computer and Information Science, University of Ljubljana. For the publication or exploitation of the master's thesis results, a written consent of the author, the Faculty of Computer and Information Science, and the supervisor is necessary.

©2018 Amela Špica

Acknowledgments

Primarily, I would like to express my gratitude to my supervisor, Prof. Andrej Brodnik for his help, guidance and patience while working on the thesis. I enjoyed dealing and learning about this topic, and our brainstorming sessions contributed in the process of solving the problem. I would also like to thank Matevž Jakovec for his unselfish help and contribution.

Moreover, I would like to thank the professors and the staff at the Faculty of Computer and Information Science, the company Comtrade and my colleagues, for providing me support in these two years of my studies as a foreign student.

Special thanks goes to my parents and my brother for their caring and their unconditional support in pursuing my career in science. They raised me to enjoy learning and eagerly explore new ideas and were always giving their maximum to pave the way for my professional development to becoming a scientist.

Finally, I am grateful to all my friends who were always there for me, backing me in every situation and providing help whenever I needed it. They made this challenging journey much easier and enjoyable.

Amela Špica, 2018

"If the doors of perception were cleansed everything would appear to man as it is, infinite."

— William Blake

Contents

Povzetek

Abstract

Razširjeni povzetek **i**

I Kratek pregled sorodnih del i
II Definicija problema ii
III Predlagane rešitve ii
IV Eksperimentalno ovrednotenje iii
V Sklep iii

1 Introduction **1**

2 Related work **3**

3 Basis and background **7**

3.1 Approximate string matching 7
3.2 Suffix tree 9
3.3 Musical terms 11
3.4 Programming tools and environment 12

4 Problem definition and modelling **13**

4.1 Musical score as a string representation 13
4.2 Formal definition 14

5 Solutions **17**
5.1 Finding the position of the soloist in the score 18
5.2 Naïve approach 20
5.3 Reward based approach 27
5.4 Reward based approach with Kalman filter 36

6 Empirical evaluation and comparison **37**
6.1 Comparison 37
6.2 Complexity 49

7 Conclusion **51**

List of used acronmys

acronym	**meaning**
HMM	Hidden Markov Model
DTW	Dynamic Time Warping
MIR	Music Information Retrieval
MIREX	Music Information Retrieval Evaluation eXchange
IRCAM	Institut de Recherche et Coordination Acoustique/Musique

Povzetek

Naslov: Računalniško podprta glasbena spremljava

Sledenje partituri je problem, pri katerem želimo v stvarnem času poizkati mesto v notnem zapisu, ki se najbolje ujema s trenutno izvedenimi toni glasbenika. V magistrski nalogi smo problem prevedli na problem približnega iskanja niza v besedilu. Za temeljno podatkovno strukturo smo uporabili priponsko drevo nad višinami not skladbe. Drevo smo razširili tako, da omogoča učinkovito približno iskanje niza z možnimi dodanimi, zamenjanimi ali izpuščenimi notami. Zasnovali, primerjali in ovrednotili smo tri pristope tovrstnega preiskovanja priponskega drevesa.

Ključne besede

korepetitor, sledenje partituri, približno iskanje niza, priponsko drevo

Abstract

Title: Computer Supported Play-along Music

Score following problem is the problem of a real-time matching of musical performance to the corresponding musical score. We also want to provide an algorithm which would follow the soloist, even when (s)he makes jumps in the score while performing. Using a string representation of the score, and an augmented suffix tree data structure built on that string, we are approaching this problem as solving an incremental approximate string matching problem. This will allow error of insertion, replacement, and deletion. We proposed three different approaches for the solution, and gave their comparison and evaluation. Besides, our solution supports soloist following even if (s)he jumps to completely arbitrary part of the composition.

Keywords

musical accompanist, score following, approximate string matching, suffix tree

Razširjeni povzetek

Sledenje partituri je problem, pri katerem želimo v stvarnem času poiskati mesto v notnem zapisu, ki se najbolje ujema s trenutno izvedenimi glasbenikovimi toni [1]. Problem ima veliko praktično uporabnost, npr. pri glasbeni spremljavi izvajalca, pri ocenjevanju kakovosti izvedbe, za samodejno obračanje strani notnega zapisa ipd. Glasbeniki običajno nimajo na voljo korepetitorja ali glasbene spremljave, oz. jim je ta na voljo zelo kratek čas (npr. enkrat tedensko v glasbenih šolah) [2].

I Kratek pregled sorodnih del

Roger Dannenberg je v članku [3] problem razdelil na tri podprobleme: 1) ugotoviti, kdaj glasbenik igra, 2) uskladiti zaznan vhod z notnim zapisom in 3) zaigrati glasbeno spremljavo. Poudarek njegovega članka je bil na problemu 2, ki ga je reševal z dinamičnim programiranjem. V naši nalogi rešujemo isti problem z uporabo približnega iskanja nizov v besedilu.

V istem obdobju je Barry Vercoe v [4] reševal problem sledenja glasbeniku s tehniko iskanja vzorcev. Postopek je kasneje izboljšal s pomočjo nadzorovanega strojnega učenja, kjer je utežil različne tipe napak glede na pričakovano nadaljevanje [5]. Poleg omenjenega je bilo kasneje razvitih nekaj podobnih pristopov z uporabo iskanja nizov v besedilu [6, 7].

Opisani problem je možno rešiti tudi z uporabo statističnih metod. Najbolj razširjene so skrite markovske verige, prvič predlagane v [8]. Uspešnost metode je primerljiva z Dannenbergovo metodo dinamičnega programiranja

predstavljeni na konferenci MIREX [9]. S področja digitalnega procesiranja signalov je znana tudi tehnika dinamičnega izkrivljanja časa [10, 11], prvotno namenjena čim boljšemu prileganju časovnih vrst.

II Definicija problema

Rešujemo problem *inkrementalnega približnega iskanja niza.* Naj T predstavlja skladbo, zapisano kot niz znakov (npr. niz tonskih višin), in P niz tonskih višin, ki jih glasbenik zaigra. Niz P lahko vsebuje tri vrste napak: dodatno zaigrana nota, napačno zaigrana nota ali izpuščena nota. Rešujemo problem iskanja mesta v T, ki se najbolje "prilega" vzorcu P. Problem je podoben problemu približnega iskanja niza P v T z najmanjšo urejevalno razdaljo, le da naš problem vsebuje tudi časovno komponento. Problem formalno definiramo kot:

Definition 0.1 *Naj Σ označuje abecedo s $\sigma = |\Sigma|$ znaki in $T \in \Sigma^*$ skladbo, predstavljeno kot niz, dolg $m = |T|$ znakov. $P \in \Sigma^*$ označuje zaigrane tone glasbenika, predstavljene kot niz, kjer je v času $0 < t \leq n = |P|$ zaigran ton $P[t]$ in kjer so najverjetneje zaigrani toni v P toni skladbe T. Algoritem naj glede na T in zaigrane tone do časa t, $P[1..t-1]$, napove naslednji zaigran ton v trenutku t, $T[i]$, tako da velja $P[t] = T[j]$. Za rešitev problema* inkrementalnega približnega iskanja niza *moramo minimizirati razdaljo med napovedanimi in dejansko zaigranimi toni*

$$d = \min(\sum_{t=1}^{n}[(i-j) \neq 0]) \ , \tag{1}$$

kjer velja $[true] = 1$ in $[false] = 0$.

III Predlagane rešitve

V nalogi predlagamo tri pristope. Pri vseh pristopih najprej zgradimo priponsko drevo za niz T. Nato preiskujemo priponsko drevo tako, da se za vsak

prebran znak v P spustimo v ustrezno poddrevo. Če poddrevesa z iskanim znakom ni, se vrnemo v koren in nadaljujemo z iskanjem naslednjega znaka v P. Vozlišče, v katerem končamo z iskanjem, hrani pozicijo v besedilu, kjer naj bi se glasbenik nahajal. Pri prvem pristopu se šešteje število pravilnih notah in tako napovedujemo položaj solista v T.

Pri ovrednotenju prvega pristopa smo pogosto opazili nezvezne skoke v notnem zapisu v primeru napake v P. Pristop smo nadgradili z nagrajevanjem vozlišč v poddrevesih, ki se začnejo s podnizom v P. Prvo vozlišče z najvišjo nagrado, ki se nahaja v T na mestu i, je izbrano za položaj v skladbi ne glede na trenutno končno vozlišče.

Pri tretjem pristopu smo vpeljali časovno komponento. Nagrado vozlišč smo dodatno utežili s Kalmanovim filtrom tako, da smo nagrado starejših vozlišč zmanjšali. "Starejšim" vozliščem, ki so bila obiskana pri preiskovanju začetka P, smo s tem znižali prioriteto.

IV Eksperimentalno ovrednotenje

Merjenje uspešnosti algoritma smo preizkusili na treh vrstah napak v P: vstavljanje, zamenjava in brisanje tona. Vhodno besedilo T je bilo velikosti $m = 681$, in smo zgenerirali P z uporabo T. Tudi primer skokov v skladbi je bil preizkušen za vsako operacijo. Vzorec P, pri skokih, smo zgenerirali iz T tako, da je $P = T[0 \ldots r_1]T[r_2 \ldots m]$, kjer sta r_1 in r_2 naključni spremenljivki in $100 < r_1 < 300$ in $400 < r_2 < 681$. Točnost iskanja za dodatno vstavljene tone je prikazana na grafih 6.3, 6.4 in 6.13, točnost iskanja pri zamenjanih tonih na grafih 6.7, 6.8 in 6.14 in točnost iskanja pri zbrisanih tonih na grafih 6.11, 6.12 in 6.15. Najuspešnejši je bil tretji pristop.

V Sklep

Predlagali smo tri pristope za reševanje sledenja izvajalcu v notnem zapisu. Vse pristope smo ovrednotili na podnizu pravilnega besedila z daljšim

izpuščenim odsekom skladbe in z manjšim številom dodatno vstavljenih, zamenjanih ali zbrisanih tonov. Pristopi so bili sprogramirani v programskem jeziku C++. Opazili smo naslednje:

1. V splošnem sta drugi in tretji pristop robustnejša od prvega pristopa.
2. Prvi pristop deluje boljše, če se napake v zaigranem odseku skladbe dovolj narazen.
3. V primeru majhnega števila napak se drugi pristop odreže bolje od prvega. V primeru več napak, ki so blizu, potrebuje drugi pristop dlje časa v zaigranem delu, da se ponovno najde.
4. Vsi trije algoritmi delujejo slabše s ponavljajočim se besedilom. Problem izvira iz oblike priponskega drevesa, saj imajo ponavljajoča se besedila globlja poddrevesa in posledično preiskovanje potrebuje daljši vzorec, da omeji število možnih pozicij v besedilu.
5. Vsi pristopi delujejo učinkovito v primeru, da izvajalec naredi nenapovedan skok v skladbi (npr. ponavljanje posameznega odseka skladbe pri vadbi).
6. Najuspešnejši pristop je tretji z uporabo Kalmanovega filtra.

Poudarek pri izvedbi vseh treh pristopov je bil na pravilnosti delovanja. Možne so predvsem izboljšave pri porabi pomnilnika. Ker je bilo v našem primeru besedilo sorazmerno kratko, nismo opazili težav med testiranjem. Pri obsežnejših skladbah skladbah pa bi bil potreben učinkovitejši zapis posameznega vozlišča in ključnih informacij v njem.

Chapter 1

Introduction

Score following is a process of real-time matching of musical performance to the corresponding score [1]. It has a wide range of applications, including a music accompaniment or score-page turning system. The motivation of this thesis is creating a computer supported play-along music system. Many musicians are experiencing a problem of the lack of an available music accompanist when needed [2]. Usually, the solution is to use a recorded accompaniment, the so-called "play-along music". It also provides an alternative for rehearsals and the actual accompanist does not have to be present all the time.

One of the first to address this problem was Roger Dannenberg in the paper "An On-line Algorithm for Real-time Accompaniment"[3]. The problem is divided into three sub-problems. First problem is to detect what is the soloist doing. Second problem is to match the detected input against the score, and third one is to produce an accompaniment. The focus of the paper was the second sub-problem, the problem of following a live performer while allowing mistakes in the performance. This problem will also be the topic of master thesis, and we will present approaches using approximate string matching.

In Chapter 2 will be described the related work. Theoretical background is given in Chapter 3 for the purposes of understanding the paper. The ap-

proximate string matching problem definition is presented, together with the definition of a suffix tree and basic musical terms. Chapter 4 gives a musical score string representation and the formal definition of the problem. In the Chapter 5, three approaches are given as solutions, including the pseudocode and an example for every solution. Finally, these three algorithms are compared and evaluated in Chapter 6 and the conclusion is given in the Chapter 7.

Chapter 2

Related work

In 1984, as it is already mentioned, Dannenberg introduced the problem of matching the detected input against a musical score. The problem was solved by using a dynamic programming algorithm which introduced a matrix containing lengths of the best match-up between the detected performance and the score. The solution also handles the errors in the performance.

That same year, Barry Vercoe [4] gave the solution to the same problem using a pattern matching techniques on pitch and time information, jointly mapped onto elements of the score. Pitch information is used as a main source, and as an addition, flute fingering information is used. By doing this, the list of possible sounding pitches is reduced to three elements, as the pitch detection alone was difficult to catch and parse. Later in 1985, this solution is improved by introducing learning techniques [5]. For this solution the score following is done by assigning weights to different events of a missing note, an extra note played or a wrong note played. It calculates the sum of all weights and finds the least costly theory of events.

In the 1990s, Puckette and Lippe [6] gave the solution of keeping the pointers to prior notes which have not been matched, named a skip list. If the notes from the skip list cannot be matched, the algorithm continues the search from the current note. Weakness of the algorithm is that the composers were often forced to make compromises for ensuring the score

follower to follow the performance.

Furthermore, in 2001, Pardo and Birmingham [7] proposed an algorithm which generates sequence of chords from the performance by removing structurally unimportant notes. Later, this sequence of chords was aligned against the sequence of chords in the lead sheet which was known in advance. Dynamic programming algorithm was used to find an optimal alignment between two sequences to determine the most optimal place in the lead sheet.

Algorithms described so far were based on string matching techniques. In this thesis, we will also introduce solutions based on the string matching algorithms. The implementation of the proposed algorithms is built on a suffix tree data structure which will easily enable the soloist to start playing from any part of the score. There are other approaches for solving the score following problem that will be described, besides already mentioned solutions.

After solutions based on string matching techniques and some of them combined with statistics, a new approach was proposed using Hidden Markov Models (HMM). A drawback of the algorithms using HMM is that they need to be trained in order to perform well, since this method is based on learning from past experiences. One of the most famous versions using HMM is the IRCAM score following algorithm, first described in 2003 [8]. Later, in 2006, this algorithm was evaluated on its precision for matching events in the audio to the score at the Music Information Retrieval Evaluation eXchange (MIREX) [9]. MIREX is a conference for evaluation of the music information retrieval (MIR) algorithms. Two algorithms were compared, and the IRCAM's algorithm outperformed the version of Dannenberg's 1984 dynamic programming algorithm.

Lastly, the approach based on technique for aligning time series, called dynamic time warping (DTW), will be mentioned. Nicola Orio and Diemo Schwarz [10] were the first ones to propose this solution. In their solution, sequences to be aligned are frames consisting of features. The feature data for the performance is extracted by signal analysis techniques. Implementation of this algorithm is consuming a lot of memory because of the matrices which

need to be stored. An example from the paper says that Sonata 1 for solo violin by J.S.Bach from Bach-Werke-Verzeichnis catalogue of compositions, which contains 450 notes, produces about 24 000 frames for the performance and the score, and matrices of around $0.5 \cdot 10^9$ elements, which are taking up at least 2 GB of memory. By introducing constraints on the frame, where just the first and the last score frame for each note is taken, memory requirements can be lowered down to approximately 10^7 elements, i.e. 40MB. Later, there were efforts in order to decrease the complexity.

In the paper [11] a framework based on spectral factorization and DTW is presented. The framework has two stages: preprocessing and alignment. Alignment between a score and an audio performance is done using the DTW based method on the cost matrix. The cost matrix is computed using the divergence matrix obtained by using a non-iterative signal decomposition method. Finally, the alignment is performed in online and offline manner. Obtained results from the online alignment are degraded comparing to the results from the offline alignment. The algorithms based on DTW are different in its approach to the problem that we are trying to solve. It is based on listening to the music and aligning audio frames, whereas our approach to the problem is based on the string matching techniques.

Chapter 3

Basis and background

String matching problem is a widely present problem in computer science, whether it is finding a specific word in a text, natural language processing, matching a regex pattern or using it for a DNA sequence pattern matching in bioinformatics. Definition of the exact string matching is given in the Dan Gusfield's book for the string matching algorithms [12].

Definition 3.1 *Given a string P called the pattern and a longer string T called text, the exact string matching problem is to find all occurrences, if any, of pattern P in text T.*

3.1 Approximate string matching

The focus of this thesis will be approximate string matching, which is a string matching problem that allows errors. The problem is to find a pattern in a given text, with errors in matches [13]. Formally, approximate string matching is defined as:

Definition 3.2 *Let Σ be a finite alphabet of size $\sigma = |\Sigma|$, and $T \in \Sigma^*$ be a text of length $m = |T|$. Further, let $P \in \Sigma^*$ be a pattern of length $n = |P|$, and $k \in \mathbb{R}$ be the maximum error allowed. Finally, let $d : \Sigma^* \times \Sigma^* \rightarrow \mathbb{R}$ be a distance function. The problem is: given T, P, k and $d(\cdot)$, return the set of all the text positions j such that there exists i such that $d(P, T_{i..j}) \leq k$.*

For the number of errors and for deciding how different two strings are, there are different approaches. They can be presented by distance, which is formally defined as follows [13]:

Definition 3.3 *The distance $d(x, y)$ between two strings x and y is the minimal cost of a sequence of operations which transform x into y. Sum of each operation cost is the distance. The operations are a finite set of rules of the form $\delta(z, w) = t$, where z and w are different strings and t is a non-negative real number. Once the operation has converted a substring z into w, no further operations needs to be done on w.*

In most cases these operations are [13]:

Insertion: $\delta(\epsilon, a)$, which means letter a is inserted in the string ϵ.

Deletion: $\delta(a, \epsilon)$, which means letter a is deleted from the string ϵ.

Substitution or replacement: $\delta(a, b)$ for $a \neq b$, which means substitution of letter a by b.

Transposition: $\delta(ab, ba)$ for $a \neq b$, which means swap the adjacent letters a and b.

Given these operations, the best-known distances are:

Levenshtein or edit distance [14] which is the most commonly used approach. It allows operation of insertion, deletion, and replacement. Edit distance for which all operations have cost 1 is called simple edit distance, and it is mostly used for approximate string matching.

Hamming distance [15] which allows only substitutions of cost 1, also known as "string matching with k mismatches".

Episode distance [16] which allows only insertions of cost 1, also known as "episode matching".

Longest common subsequence distance [17] which allows only insertions and deletions of cost 1.

Operations from the edit distance will be used in this thesis, so a simple explanation of calculating this distance will be given. For two strings x = `abcde` and y = `abbe`, the edit distance is $ed(\texttt{abcde}, \texttt{abbe}) = 2$. This means that two operations are required in order to get one string from the other: (i) substitution – letters c to b; and (ii) deletion – the letter d.

3.2 Suffix tree

Suffix tree [18] is a data structure used for finding suffixes in a text. It solves the exact string matching problem in a linear time. Given a text T of length m, and a pattern P of length n, the suffix tree τ for the text is built in $O(m)$ time during preprocessing phase, and search of the pattern P in the suffix tree takes $O(n)$ time. Formally, it is defined in the Definition 3.4 (Definition 5.2 in [12]).

Definition 3.4 *A suffix tree τ for an m-character string T is a rooted directed tree with exactly m leaves numbered from 1 to m. Each internal node has at least two children and each edge is labeled with a nonempty substring of T, called edge-label. No two edges coming out from the same node can have the same first character of edge-label. For any leaf i, concatenation of edge-labels from root to node i represents substring $T[i...m]$.*

Weiner was the first one to give a linear-time algorithm for the construction of the suffix tree. After that, McCreight [19] gave more space efficient algorithm. Later, Ukkonen [20] developed an algorithm which is also used in this thesis as a suffix tree build, with our augmentation to fit the algorithm which will be described later. The whole process of building a suffix tree will not be described, but a simple example will be given for easier understanding.

Given the text T = `abcabaxabcd`, the suffix tree will look as in the Figure 3.1. Usually, every string ends with the character $ but we will omit it for now.

The tree contains internal nodes labeled with the capital letters: A, B, C, D, E, and F. The first node is a root node. Nodes containing numbers

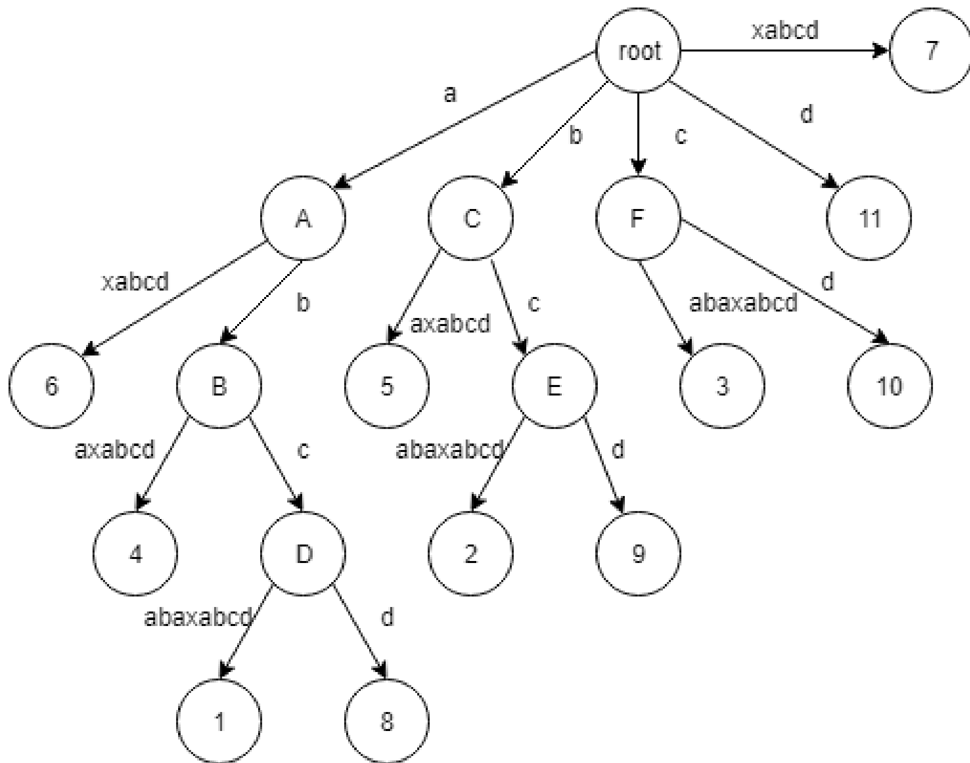

Figure 3.1: Example of the suffix tree for text $T =$ `abcabaxabcd`.

as node labels are leaves. The number on the leaf represents suffix index of that leaf. Concatenation of edge-labels on the path from the root to the leaf i gives the suffix of T that starts at position i, i.e. $T[i...m]$. For example, for the substring $P =$ `abcd`, the path while walking down the tree is: $A \rightarrow B \rightarrow D \rightarrow 8$. The starting position of the substring P is 8, that is, its suffix index is 8.

3.2.1 Data structure

In order to explain the algorithm and pseudocode, it is necessary to introduce the structure of the suffix tree that was used. Suffix tree has nodes, which contain information about the node and pointers to the children of the node. Furthermore, every node has information about the first and the last

position of the substring they are representing. For example, for the text T = `abcabaxabcd` the edge which represents the substring P = `abcd` will have the following information: $node.first = 8$ and $node.last = 12$. This means that P is the substring starting from the position 8 and ending at the position 12.

We augment a usual suffix tree in such a way that each internal node n contains also a list of all suffix indices at leaves of a subtree rooted at n. Given the text T = `abcabaxabcd` suffix tree representation with suffix indices looks as in the picture 3.2. Labels on the edges are representing suffix indices of the internal node. For example, suffix indices for the node 8 are 1, 4, 6, 8. This means that the letter a from the text T can be found at the positions: $1, 4, 6, 8$.

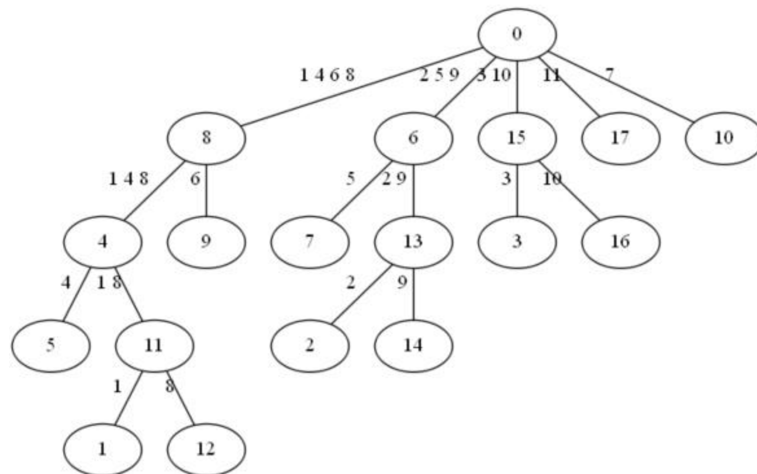

Figure 3.2: Suffix tree representation of the score with string representation T = `abcabaxabcd`.

3.3 Musical terms

Before explaining the algorithms, terms used for the definition of the problem will be formally defined [21].

Musical score: Notation which is used for representing notes for playing. The musical score will be presented as a string T, which will be thoroughly explained in Chapter 4.

Performance: In this context, performance is the situation when a solo musician (soloist) is performing a piece of music. This musician can be for example a flute player or a singer. The soloist is then accompanied by another musician (accompanist), which can be, for example, pianist or even a whole orchestra. The goal of the accompanist is to try to follow the soloist, regardless of the starting position of the soloist.

Soloist: The solo musician who is performing the piece. The part which soloist plays is represented as a pattern P which is being matched against the score string representation T.

Accompaniment: The musician who is playing the accompaniment. (S)he is following the soloist performance.

Score follower: A computer accompanist that follows the soloist through the performance. It is matching the soloist performance to the score and playing the accompanist part of the score.

3.4 Programming tools and environment

Implementation was done using a programming language C++ and code is publicly available on Github repository. For the graph representation we used Graphviz software. The code was run on the Windows 10 operating system and MinGW compiler was used with default settings inside of CLion IDE.

Chapter 4

Problem definition and modelling

The problem that we are solving is score following, the problem of matching a real-time musical performance to the corresponding score. This means that the proposed algorithm will try to evaluate and follow the part which is being played at the moment. The algorithm can be used for the multiple purposes, such as the automatic page turner for the player or for implementing a computer accompanist which will play accompanist part of the player score.

4.1 Musical score as a string representation

A musical score can contain a lot of information, but we will focus just on the notes. They will be represented as a string, in order to simplify and transform the problem to the string matching problem.

The score is presented in a form of a string. Every note has information about three following note properties: tone, octave, and sign. A tone is defined from the set: c, d, e, f, g, a, b. Octave is any number between -1 and 10. A sign can be natural, sharp and flat, which is represented in the string with characters $\natural$, $\sharp$, and $\flat$, respectively. For example, notes given in the

Figure 4.1 will have the following string representation: c2♮e3♯a2♮d3♭. For the purpose of simplicity duration and other musical properties are omitted.

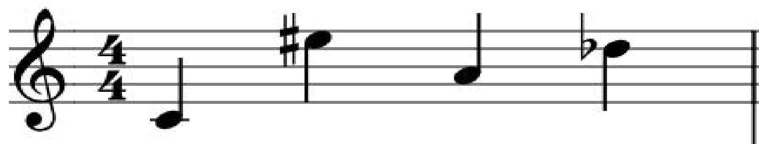

Figure 4.1: Score example.

In the current representation, three characters are representing one note. If we denote the string representation as string S, each triplet in S can be mapped into one character, which will form the new score string representation T.

Having the score in a string form allows a suffix tree to be built from this string. This permits the score following process to start from any position in the score and to continue following the notes by walking down the tree.

4.2 Formal definition

The problem has been reduced to what we define as *a problem of incremental approximate string matching*. Informally, we have a music score (text) $T = a_1, a_2, ..., a_m$ that is played by the soloist. If the soloist would make no mistakes, it produces the pattern $P = T$. However, since the soloist makes mistakes the produced music $P = b_1, b_2, ..., b_n$ is different from T. That is, the soloist at time $t = 1, 2, ..., n$ plays note b_t, while it *intends* to play note a_i. Observe that t is incremented by 1 in every step, while i can (in general) jump arbitrary. However, due to the nature of the problem, most probably also i is incremented by 1.

In general, a soloist makes three kinds of mistakes: (i) plays a different note than a note in a score (substitution) – both t and i are incremented by 1; (ii) misses a note from the score (deletion) – t is incremented by one, while i is incremented by 2; or (iii) plays an additional note that is not in the score

(insertion) – t is incremented by 1, while i remains unchanged. Furthermore, we can also model a situation when the soloist skips a number of bars, or repeats a piece of music as a sequence of mistakes.

The problem we want to solve is to design an algorithm, that based on the produced notes $b_1, b_2, ..., b_{t-1}$ and a music score T, predicts what is the next note b_t produced by the soloist. Obviously, if the produced note b_t is a_i and our algorithm predicted to be a_j, we want $(i - j) = 0$.

More formally we define a problem of incremental approximate string matching as:

Definition 4.1 *Let Σ be a finite alphabet of size $\sigma = |\Sigma|$, and $T \in \Sigma^*$ be a string representation of the musical score of length $m = |T|$. Further, let $P \in \Sigma^*$ be a string representation of notes played by the soloist, where at the time $0 < t \leq n = |P|$ is played note $P[t]$ and most probably consecutive played notes in P are also consecutive notes in a score T. Finally, let the algorithm based on score T and notes produced up to time t, $P[1..t-1]$ make the prediction that the soloist will play at the time t note $T[j]$ and the soloist plays note $P[t] = T[i]$. To solve* the problem of incremental approximate string matching *we have to minimize the distance between the predictions and played notes*

$$d = \min(\sum_{t=1}^{n} [(i - j) \neq 0]) \ , \tag{4.1}$$

where $[true] = 1$ and $[false] = 0$.

For the clarification of the definition we give the following example (see also Table 4.1). We are given the text $T =$ `cdcbcdecdcba` $(m = |T| = 12)$ and the pattern $P =$ `cdcbcdecadcba` $(n = |P| = 11)$. In this case, there is one insertion after the position $t = 8$. Table 4.1 shows for each $t \in [1, n]$ a position i, that is the positions of note intended to be played by the soloist, and position j, that is the positions of note in the score T suggested by the algorithm.

In the example presented in Table 4.1, the mistake happened at the time $t = 9$ for the intended position $i = 9$ and the suggested position was $j = 12$.

t	1	2	3	4	5	6	7	8	9	10	11	12	13
P	`c`	`d`	`c`	`b`	`c`	`d`	`e`	`c`	`a`	`d`	`c`	`b`	`a`
T	c	d	c	b	c	d	e	c		d	c	b	a
i	1	2	3	4	5	6	7	8	9	9	10	11	12
j	1	2	3	4	5	6	7	8	12	9	10	11	12

Table 4.1: Played notes P with indices into T for the soloist and as predicted by the algorithm.

Consequently we get a distance $d = 1$.

Chapter 5

Solutions

Descriptions of the proposed solutions to the mentioned problem will be given in this chapter. As it is already specified, we are trying to provide a fast solution for handling the problem of incremental approximate string matching, different from calculating edit distance with dynamic programming algorithms. The dynamic program's complexity $O(n \cdot m)$ is not acceptable for us.

Given the score, for each note played by the soloist the algorithm returns its predicted position in the score. Then it predicts which note is next to be played by the soloist and plays the accompanist part for that note. As defined in Definition 4.1, most probably it will be the next note in the score. Pseudocode for this is given in the Algorithm 1.

Algorithm 1 Pseudocode for play-along

```
1: function PlayAlong(T, τ)
2:     j ← 0
3:     for t = 1; t not at the end of performance; t++ do
4:         PLAY(T[j + 1])
5:         playedNote ← LISTEN()
6:         j ← FindPositionInTheScore(playedNote, T, τ)
7:         pattern.append(playedNote)        ▷ form the pattern from played notes
8:     end for
9: end function
```

Representation of the score will be in a string format as it was explained in the Chapter 4. This will allow solving the problem as the string matching problem. Furthermore, allowing mistakes in the pattern and performing it in real-time will narrow the problem down to the incremental approximate string matching problem. Since the problem definition is also allowing the player to start playing from any part of the score, we will build a suffix tree from the score string representation, namely text T.

First, a simple approach for finding a position of the soloist in the score will be described. Later, three approaches for solving this problem will be given. In the end, these approaches will be compared given different strings to see how well they perform for different kinds of errors.

5.1 Finding the position of the soloist in the score

In the proposed solutions we are using a suffix tree data structure. We will describe the algorithm for walk down the suffix tree for each character in the pattern P which represents one note played by the soloist. The suffix tree is built on the text T, which is the string representation of the score.

While walking down the suffix tree, for each character from the pattern P, the algorithm tries to find the character on the edge to continue the walk. If the walk can be continued, it means the soloist's performance is matched to the score and that the soloist has not made a mistake. If the character from the pattern cannot be found on the edge, it means that an error in the soloist's performance occurred. The walk cannot be continued, and the algorithm returns back the root node. The walk down the tree is continued from the root, using the same character that could not be found in the previous step. If the character cannot be found on the edge, even from the root, it means that the character is not in the alphabet of the text T. In terms of performance, this means that the soloist made a mistake by playing the note that is not in the score at all.

The walk down the tree, for each character from the pattern P, is done in the following way. First, take the character from P and try to find it on the edge of the suffix tree. Since each node has a list of children, the edge has to be found from that list. As we already know, each edge of the suffix tree represents some substring of the text T. For the first iteration, algorithm starts from the root and tries to find the edge which also starts with the first character from P. More generally, the search is done by finding an edge from the children of internal node.

Any character from the text T in the suffix tree τ can be described with information about the internal node and edge offset from that node. In the algorithm, the proposed position in T is stored using mentioned information. If the edge offset is zero it means the algorithm is walking down the suffix tree from an internal node. In this case, the walk down the tree is done by finding a child which first character is the same as the current character in P. On the other hand, if the edge offset is not zero it means the algorithm is searching for the current character in P from the edge visited in the previous iteration of the walk down. In this case, walk down the tree means going to the next character on the edge and increasing the edge offset. If the next character is also the last character on that edge, edge offset is set back to zero.

After the character from the pattern P and the character from the text T are compared, the algorithm can go two ways. The first way is to continue the walk, which means that the match is found and that the soloist played the note from the score. The second way is to go back to the root, which means the character from the pattern P cannot be found on the edge.

There are different ways of calculating the suggested position j in the score T for each character $P[t]$ at the time t. We proposed three approaches, which will now be described.

5.2 Naïve approach

As it is already mentioned, each internal node has a list of suffix indices. Each leaf of the suffix tree contains information about the suffix index. Let us remind that a suffix index is a starting position of the substring of the text T that is a concatenation of the edge labels from the root to the leaf (see Figure 3.1). The list of suffix indices contains the information about each suffix index from the leaves of this internal node subtree. While walking down the tree, the algorithm suggests the position j in the text T for the character from the pattern P. In the naïve approach, this is done by taking the smallest suffix index from the internal node, not smaller than the number of correct notes played. If there is no such suffix index then take the last suffix index. This suffix index is also the biggest since the list of suffix indices is kept sorted.

A number of correct notes played is increased each time the algorithm continues walk down the tree. If the algorithm comes back to the root it means that there was a mistake in performance. This will not increase the number of correct notes. In the algorithm, this is determined by checking the parent node. If the parent node is root, and the current character for the walk down is not the first character in T, the number of correct notes played is not increased. In any other case, it is increased for each iteration. Pseudocode is given in the Algorithm 2. In order to perform the algorithm, information about the edge offset from an internal node and offset from the root are stored.

We can explain the algorithm using a simple example. Given a score string representation $T =$ `abcabaxabcd` and the pattern $P =$ `abdabaxabcd` we can see that the soloist made a replacement error on the third note, where (s)he played d instead of c. Now, let's see what happens if we run the algorithm. Score suffix tree representation looks as in the Figure 5.1.

Given this representation, we will go through the first four steps of the algorithm:

Algorithm 2 Pseudocode for the naïve approach

```
 1: function FindPositionInTheScore(currentCharInPattern, T, τ)
 2:     found ← false
 3:     for node in n.children do
 4:         currentCharOnEdge ← t[node.first − 1 + edgeOffset]
 5:         if cannot find currentCharInPattern from the node n then
 6:             break
 7:         end if
 8:         if currentCharOnEdge == currentCharInPattern then
 9:             suggestedPosition ← smallest suffix index from node.suffixIndices
10:             if node parent != root OR currentCharInPattern at first position then
11:                 numberOfCorrectNotes = numberOfCorrectNotes + 1
12:             end if
13:             found ← true
14:             update environment variables edgeOffset, n
15:         end if
16:     end for
17:     if !found then
18:         suggestedPosition ← last suggestedPosition + 1     ▷ from the t − 1 iteration
19:     end if
20:     return suggestedPosition
21: end function
```

1. First character a:

   Node 8 is taken for the walk down the suffix tree. The list of *suffix indices* is $1, 4, 6, 8$. Algorithm takes the smallest index and marks it as the played note, which means that the *suggested position* is 1. Increase the *number of correct notes* and *offset from the root*. *Offset from the edge* is zero because the walk down the suffix tree is continued from the next internal node.

2. Second character b:

   As it walks down the tree, the algorithm is now at the *node* number 4. The list of the suffix indices for substring *ab* is *node.suffixIndices* is $1, 4, 8$. Since one note is already played, the list is transformed

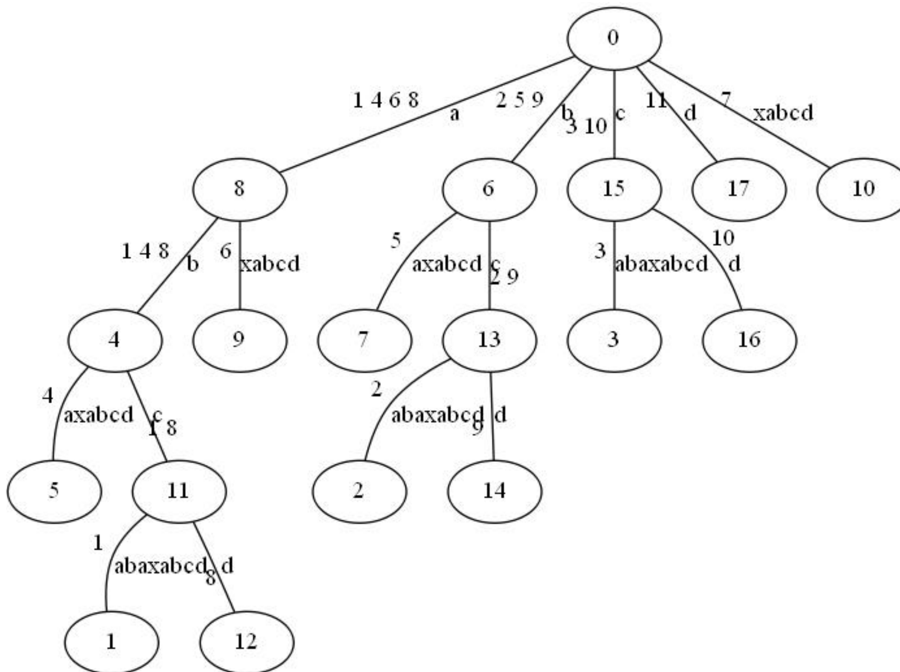

Figure 5.1: Suffix tree representation of the score T = `abcabaxabcd` with substrings and suffix indices on the edges.

by adding the *offset from the root* to the list. The list is now $2, 5, 9$ which means that these are the proposed positions for the played note. Again, the algorithm takes the smallest suffix number from the list, not smaller than the *number of correct notes* which is *position* 2. Increase the *number of correct notes* and *offset from the root*, and reset the *edge offset*.

3. Third character d:

   The algorithm cannot find any child from the *node* 4 that has an edge starting with the character d so it must search from the root again. This will reset the *offset from the root* and it will not increase the *number of correct notes*. As it starts searching from the root, it ends up in the internal node number 17, with *node.suffixIndices* = 11. Since the *offset from the root* is zero, it means that the smallest number, bigger

or equal to *number of correct notes* is 11. The algorithm will suggest that the player is at the *position* 11, even though the player is actually at the position 3.

4. Fourth character a:

   Given the char a, the algorithm cannot continue the walk from the *node* 17 so it returns to the root. Again, it ends up in the *node* 8 with *node.suffixIndices* $= 1, 4, 6, 8$. It suggests that the soloist is in the position 4 since that is the minimum number not smaller than *number of correct notes* $= 2$. From now on algorithm continues the walk down the tree.

5.2.1 Example

We will show how the algorithm was tested and what are the possible problems for each type of the operation from edit distance: insertion, replacement, and deletion. The evaluation is primarily done using strings of a small size. In the Chapter 6 we will show how the algorithm behaves on larger strings.

Insertion

In order to test insertion, we wrote a programme which adds a random note from the score at a random place. This creates the pattern P. The algorithm gets the score string representation – text T, which needs to be transformed to the pattern, and number of errors k, as arguments. Pseudocode for this is given in the Algorithm 3.

For the evaluation we first took "Twinkle Twinkle Little Star", as it was short and simple for making the observations. Score string representation of this score is $T =$ `aabbccbddeeffabbddeefbbddeefaabbccbddeeffa`\$. Notice that now we added the character \$ to denote the end of the string. We will present results in table for a given pattern and present the evaluation. The number of mistakes is $k = 3$.

Algorithm 3 Pseudocode for inserting k random notes

1: **function** INSERTION(k, T)
2:     $pattern \leftarrow$ empty string
3:     $positions \leftarrow$ generate k random numbers for positions
4:     $letters \leftarrow$ generate k random numbers for letters
5:     sort $positions$ in descending order
6:     **for** $i \leftarrow 0; i < k; i \leftarrow i + 1$ **do**
7:         $letter \leftarrow T[letters[i]]$
8:         $randNum \leftarrow positions[i]$
9:         $pattern.append(randNum, letter)$ ▷ insert $letter$ at the $randNum$ position
10:     **end for**
11:     **return** $pattern$
12: **end function**

For the example of insertion let's take the pattern $P =$ `aabbccbddeeffabbdedeefbebddeefadabbccbddeeffa`. Figure 5.2 shows the positions proposed by the naïve approach, for the part of the pattern P where insertion errors occurred. In the row for the results of naïve approach, numbers in red are the positions which are wrongly suggested. Intended positions are shown in the second row.

pattern	d	e	d	e	e	f	b	e	b	d	d	e	e	f	a	d	a	b	b
intended position	17	18	18	19	20	21	22	23	23	24	25	26	27	28	29	30	30	31	32
naive approach	17	19	17	19	20	21	22	26	22	24	25	26	27	28	29	36	29	31	32

Figure 5.2: Intended and proposed positions for the pattern $P =$ `aabbccbddeeffabbdedeefbebddeefadabbccbddeeffa`.

Replacement

In order to test replacement, we wrote a programme which replaces the character at a random position with a random letter from the string. Pseudocode is given in the algorithm 4.

For observing results we again took "Twinke Twinkle Little Star", with

Algorithm 4 Pseudocode for replacing k random notes

1: **function** REPLACEMENT(k, T)
2:     $pattern \leftarrow$ empty string
3:     $positions \leftarrow$ generate k random numbers for positions
4:     $letters \leftarrow$ generate k random numbers for letters
5:     sort *positions* in descending order
6:     **for** $i \leftarrow 0$; $i < k$; $i \leftarrow i + 1$ **do**
7:         $letter \leftarrow T[letters[i]]$
8:         $randNum \leftarrow positions[i]$
9:         $pattern.replace(randNum, letter)$ ▷ replace the character at the $randNum$ position with $letter$
10:     **end for**
11:     **return** *pattern*
12: **end function**

score string representation $T =$ `aabbccbddeeffabbddeefbbddeefaabbccbdd eeffa$`. For the pattern $P =$ `aabeccbddeefffbbddbefbbddeefaabbccbddee ffa` part of the results with replacement errors are given in the Figure 5.3.

	a	a	b	e	c	c	b	d	d	e	e	f	f	f	b	b	d	d	b	e	f
intended	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
naive approach	1	2	3	10	5	6	7	8	9	10	11	12	13	13	22	23	24	25	22	19	21

Figure 5.3: Intended and proposed positions for the pattern $P =$ `aabeccbddeefffbbddbefbbddeefaabbccbddeeffa`.

Deletion

The deletion operation is similar to the operation of insertion. The algorithm has to skip one note from the score. We tested this again against the same text T and results are given in the Figure 5.4 for the pattern $P =$ `aabbccbddeeffabbddefbbddeaabbccbddeeffa`.

Pseudocode for forming the pattern for deletion operation is given in the Algorithm 5.

	d	d	e		f	b	b	d	d	e			a	a	b
intended	17	18	19		21	22	23	24	25	26			29	30	31
naive approach	17	18	19		21	22	23	24	25	26			29	30	31

Figure 5.4: Intended and proposed positions for the pattern P = `aabbccbddeeffabbddefbbddeaabbccbddeeffa`.

Algorithm 5 Pseudocode for deleting k random notes

```
 1: function DELETION(k, T)
 2:     pattern ← empty string
 3:     positions ← generate k random numbers for positions
 4:     sort positions in descending order
 5:     for i ← 0; i < k; i ← i + 1 do
 6:         randNum ← positions[i]
 7:         pattern.delete(randNum)      ▷ delete the character at the randNum position
 8:     end for
 9:     return pattern
10: end function
```

Observation

The naïve approach performs well for the mentioned examples. It is observed that, even though the algorithm performs well on short strings, it has a problem with the larger strings, which will be shown in the next Chapter 6. Handling errors by starting the walk down the tree again from the root, makes the algorithm jump too much between the values of positions for each mistake. This causes too big differences between proposed positions that are close to each other in the pattern. This happens because it is difficult to properly count the number of correctly played notes.

We are doing it by increasing the number of correct notes each time the algorithm walks down the tree, and the parent node is not root, except for the first character of the pattern P. The fact is that if one mistake happens (for example insertion) the algorithm will go back to the root two times. First time will be for the wrong note played and the second time will be for

the next note in order to get back on the track. This means that the second time, even though the note is correctly played, it will not increase the number of correct notes as the walk down the tree is again starting from the root.

The four steps we explained represent the case in which such a problem occurred. Recall that in the third step the algorithm returned to the root node. This is good, because the played note was a mistake. But also, the algorithm return back to the root for the fourth step, where we know the note was correctly played. This marks two notes as wrongly played, instead of just one.

When multiple number of mistakes like this happen, the number of correct notes differ too much from the actual number of correctly played notes. This is making a big problem when choosing a suffix index not smaller than the number of correct notes. Noticing these flaws in the algorithm, made us propose another version. It will make these jumps between proposed characters, which are relatively close to each other, more difficult.

One thing has to be pointed out. Even though we do not want big jumps between relatively close positions, we also should not discourage the algorithm from making big jumps. We already said that we want the solution to have an ability to follow the player, even when the soloist is making a big jumps in the score.

5.3 Reward based approach

In order to improve the algorithm, we introduced the second solution. It will keep the track of the positions which are possible candidates for the next character. This information will be kept in the table of rewards of each position in the string. It will enable fast recovery from an error and it will make big jumps between positions more difficult.

The algorithm uses the same way of walking down the tree as the naïve approach. The only difference is a table of rewards which stores a number of rewards for each suggested position during the walk down. It will propose

a possible candidates for the suggested position in the score T of the played note. The table has two types of information: position and reward. The position will be a suggested position in the string T of the found character from the pattern P. The reward is calculated as the sum of appearances of each position encountered as the suffix index while walking down the tree. The suffix index is found from the list of suffix indices for every internal node visited. For each appearance, the reward is increased by 1.

The most important part about reward based approach is how it handles mistakes. As it is already said, there are three types of mistakes: insertion, deletion, and replacement. On a simple example, it will be shown how these errors are solved. Let's say while playing the score, the soloist makes a mistake at the fifth note. In the case that mistake is an insertion, it means that the note on the position 5 in the pattern P is extra. Furthermore, note that is played on the sixth place is the one which should have been played on the fifth place, and the one from seventh place should have been played on the sixth place, and so on. Information about time (t) and intended position in the text (i), from the formal definition 4.1, is shown in the Table 5.1.

t	1	2	3	4	5	6	7
i	1	2	3	4	5	5	6

Table 5.1: Table for reward based approach if insertion happens at 5th place.

On the other hand, if the mistake is a replacement, and if it is again at the fifth place, it means that the accompanist intended note was the fifth note, and the sixth played note is the sixth note from the text T, etc. Intended positions are shown in the Table 5.2.

Lastly, if the error is deletion, it means that the fifth note is skipped and that the soloist jumped to the sixth note immediately, and played it fifth instead of sixth. Intended positions are shown in the Table 5.3.

Observing these tables one conclusion can be made. If a mistake happened

t	1	2	3	4	5	6	7
i	1	2	3	4	5	6	7

Table 5.2: Table for reward based approach if replacement happens at 5th place.

t	1	2	3	4	5	6
i	1	2	3	4	6	7

Table 5.3: Table for reward based approach if deletion happens at 5th place.

at the x-1 position while playing the score, the candidates for the next note at the position x are $x-1$, x, or $x+1$. In this example, $x-1=5$ and the sixth note (x) could be: (i) 5 ($x-1$) – in case of insertion; (ii) 6 (x) – in case of replacement; or (iii) 7 ($x+1$) – in case of deletion. This algorithm solves it in a way that each time a mistake happens, it adds all three possible options for the position to the table. The position which is next, in this case x, will also have its reward increased. This means that default mistake is replacement, since it does not make sense to play the same note again or to skip one note. In case it is actually an insertion or deletion, the algorithm will recognize it in the next step.

The algorithm starts the same way as the first approach. First, it builds a suffix tree from the score string representation with the proposed augmentation. For each new character in the pattern, it walks down the tree and adds all positions of suffix indices to the table, and increases their reward. In case mistake happens it starts to follow from the root, but the proposed positions for handling the mistake will also be added to the table.

Pseudocode of the algorithm is presented in the Algorithm 6 and pseudocode for adding elements to the table is given in the Algorithm 7.

The approach with rewards will be described in the first four steps. Given the score string representation $T=$ `abcabaxaxabcd` and pattern $P=$

Algorithm 6 Pseudocode for the reward based approach

```
1:  function FindPositionInTheScore(currentCharInPattern, T, τ)
2:      found ← false
3:      for node in n.children do                                   ▷ node n is from τ
4:          currentCharOnEdge ← t[node.first − 1 + edgeOffset]
5:          if cannot find currentCharInPattern from the node n then
6:              break
7:          end if
8:          if currentCharOnEdge == currentCharInPattern then
9:              table = AddToTheTable(node, table, offsetFromRoot, T, lastMax)
10:             suggestedPosition ← table[0]    ▷ Take the element with maximum reward,
    which is always kept as the first element of the table
11:             found ← true
12:             update environment variables edgeOffset, n, lastMax
13:         end if
14:     end for
15:     if !found then
16:         suggestedPosition ← last suggestedPosition + 1    ▷ from the t − 1 iteration
17:     end if
18:     return suggestedPosition
19: end function
```

`abdabaxabcd`, suffix tree representation is the same as in the picture 5.1 and this approach will have following steps:

1. First character a:

   Given the character a, the algorithm starts from the *node* 0 to the *node* 8. The *offset from the root* is 1 and the list of *node.suffixIndices* $= 1, 4, 6, 8$. Each suffix is added to the *table* and reward is set to 1 (see Table 5.4).

position	1	4	6	8
reward	1	1	1	1

Table 5.4: Table for the first step of the reward based algorithm.

Algorithm 7 Pseudocode for adding elements to the table

1: **function** ADDTOTHETABLE($node$, $table$, $offsetFromRoot$, $text$, $lastMax$)
2: mark first element as max ▷ max is the element with maximum reward
3: **while** $i < table.size$ **do**
4: **if** mistake happened in the performance **then**
5: add two extra positions for handling errors
6: delete all elements from the table with same position as two added elements
7: increase reward for the replacement mistake
8: **end if**
9: $table.position \leftarrow table.position + 1$
10: **if** $table[i].position == suffix + offsetFromRoot$ **then**
11: increase reward for $table[i].reward$
12: **end if**
13: **if** ($table[i].reward > max.reward$ or ($table[i].reward == max.reward$ and $table[i].position < max.position$)) **then**
14: $max = table[i]$
15: **end if**
16: $i \leftarrow i + 1$
17: **end while**
18: $max \longleftrightarrow table[0]$ ▷ keep max always as the first element
19: add suffix indices, with 1 reward, which were not added to the $table$ in the line 10
20: $lastMax \leftarrow table[0]$
21: **return** $table$
22: **end function**

All rewards are the same so the smallest position will be taken. The *suggested position* is 1 as it is shown in the Table 5.4 with blue color. This will give the correct suggestion.

2. Second character b:

   Suffix indices for node 4 are $1, 4, 8$. Since the *offset from the root* is 1 this means that *suffix indices* are increased to 2, 5, 9, and these positions will have their *table.reward* value increased. Also, all the *positions* in the table are increased by one. The table content is as in the Table 5.5.

position	2	5	7	9
reward	2	2	1	2

Table 5.5: Table for the second step of the algorithm with rewards.

   Positions with reward's value equals 2 are considered, and the one with the smallest position is taken, meaning $table.position = 2$. This will also give the correct *suggested position*.

3. Third character d:

   When encountering a character d the algorithm cannot walk down the tree but must go back to the *root*. The *offset from the root* is re-set to 0 and the algorithm ends up in the *node* 17. For this node $node.suffixIndices = 11$, so the position 11 is added to the *table*. Since the active node is the root node, the mistake happened for the position $x = 3$. The positions for handling the mistake are: $x - 1 = 2$ – in case of an insertion; $x + 1 = 4$ – in case of a deletion, are added to the table with the same reward as for $x = 3$. The *reward* for $x = 3$ is increased so that the algorithm will always *suggest the position* after the last position suggested. The table content is as in the Table 5.6.

position	2	3	4	6	8	10	11
reward	2	3	2	2	1	2	1

Table 5.6: Table for the third step of the algorithm with rewards.

Again, the element with the biggest reward and the smallest position is taken. Even though the soloist played the wrong note, the algorithm *suggests* the correct *position* number 3.

4. Fourth character a.

From the *node* 17, the algorithm cannot continue, so it goes back to the *root* again. Given letter "a", it ends up again in the node 8 with *suffix indices* $1, 4, 6, 8$. Adding these elements to the *table* will result in having a content as in the Table 5.7.

position	1	3	4	5	6	7	8	9	11
reward	1	2	3	2	1	2	1	1	2

Table 5.7: Table for the fourth step of the algorithm with rewards

Position 4 has the biggest reward, meaning the biggest probability that the soloist is playing this part, which is correct. From now on, the algorithm performs well.

5.3.1 Example

We will show how this algorithm performs for each type of the operation from edit distance: insertion, replacement, and deletion. The evaluation is again done using string of a small size. In the Chapter 6 it is shown how the algorithm behaves on larger strings.

Insertion

For the evaluation we again took the same score “Twinkle Twinkle Little Star”. Score string representation of this score is $T =$ `aabbccbddeeffabbdd eefbbddeefaabbccbddeeffa$`. Let us take the same pattern as for the naïve solution $P =$ `aabbccbddeeffabbded eefbebddeefadabbccbddeeffa`. Figure 5.5 shows the positions proposed by the reward based approach, for the part of the pattern P where insertion errors occurred. Intended positions are shown in the second row. Numbers in red are the positions which are wrongly suggested by the reward based approach.

pattern	d	e	d	e	e	f	b	e	b	d	d	e	e	f	a	d	a	b
intended position	17	18	18	19	20	21	22	23	23	24	25	26	27	28	29	30	30	31
reward based	17	18	19	19	20	21	22	23	24	24	25	26	27	28	29	30	31	31

Figure 5.5: Intended and proposed positions by the reward based algorithm for the pattern $P =$ `aabbccbddeeffabbdedeefbebddeefadabbccbddeeffa`.

Replacement

Replacement will be shown using the same variables as for the naïve approach. Given the same text $T =$ `aabbccbddeeffabbddeefbbddeefaabbccb ddeeffa$` and the pattern $P =$ `aabeccbddeefffbbddbefbbddeefaabbccbddeeffa` part of the results with replacement errors is given in the Figure 5.6.

	a	a	b	e	c	c	b	d	d	e	e	f	f	f	b	b	d	d	b	e	f
intended	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
reward based	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Figure 5.6: Intended and proposed positions by the reward based algorithm for the pattern $P =$ `aabeccbddeefffbbddbefbbddeefaabbccbddeeffa`.

Deletion

Deletion operation is similar to the operation of insertion. The algorithm has to skip one note from the score. For insertion it has to repeat the same note. It is already explained how the algorithm handles these mistakes. What was noticed while testing the algorithm is that in case of two consecutive mistakes of insertion or deletion, the algorithm performs bad. The reason for this behavior is that in the process of adding positions to the table, it is proposed that only one mistake will happen and the next note will probably be correct. That is not always the case. When being tested for the pattern P = `aabbccbdeeffabbddeefbbddeeaabbccbddeffa` the algorithm performed even worse than the naïve solution. This is presented in the Figure 5.7, where part with two consecutively skipped notes is pointed out.

	e	e	f			d	d	e	e	f	a	a	b	b	c	c	b	d	d	e		f	f	a
intended		20	21			24	25	26	27	28	29	30	31	32	33	34	35	36	37	38		40	41	42
naive approach	19	20	21			24	25	26	27	28	29	30	31	32	33	34	35	36	37	38		40	41	42
reward based	19	20	21			22	23	24	25	26	27	28	29	30	31	32	33	34	35	36		37	38	39

Figure 5.7: Intended and proposed positions by the reward based algorithm for the pattern P = `aabbccbdeeffabbddeefbbddeeaabbccbddeffa`.

5.3.2 Observation

As it is already said, the algorithm does not take into account that two consecutive notes can be wrongly played. In this case, the algorithm proposes a new position in the table of positions with reward 1. The bigger the number of characters examined from the root to the character where the walk cannot be continued is, the more difficult it is to recover from an error. This happens because the reward is increased each time by one, and all positions have the same importance. To solve this problem, we proposed the solution which filters old positions using the Kalman filter [22].

5.4 Reward based approach with Kalman filter

The regular reward based approach had problems with assigning the same importance to old and new events. This makes the rewards' values in the table grow too high, which hardens the change of positions and big jumps in the pattern. In the last approach we suggested that old events should be filtered using Kalman filter.

This is done by multiplying rewards for each iteration with some number $0 < c < 1$. Introducing the filter will decrease the reward importance of the old elements in the table, and the new ones will have bigger values. After applying this filter, we will add one more row to show how the algorithm performs, compared to the old versions. For the following example, the formula for calculating the reward is given in the equation 5.1 and results are presented in the Figure 5.8.

$$table.reward = table.reward \cdot 0.9 \tag{5.1}$$

The pattern on which all three algorithms are compared is $P =$ `aabbccbddeeffabbddeefbddbddedefaabbccbddeeffa`.

	e	e	f	b	d	d	b	d	d	e	d	e	f	a	a	b	b	c	c	b	d	d	e	e	f	f	a
intended	19	20	21	22	23	23	23	24	25	26	27	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
naive approach	19	20	21	22	24	25	23	24	25	26	36	26	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
reward based	19	20	21	22	23	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	43	43	42
reward based with filter	19	20	21	22	23	25	26	27	28	29	30	31	32	33	34	35	32	33	34	35	36	37	38	39	40	41	42

Figure 5.8: Intended and proposed positions from all three algorithms for the pattern $P =$ `aabbccbddeeffabbddeefbddbddedefaabbccbddeeffa`.

Chapter 6

Empirical evaluation and comparison

After introducing three approaches as the solution to the problem, the discussion regarding edit distance operations in approximate string matching will be introduced. These operations are insertion, deletion, and substitution. We will consider each operation separately and compare them. After that, the complexity of these algorithms will be described.

6.1 Comparison

In the following section, we will describe how the algorithm performs for each operation of edit distance: insertion, replacement, and deletion. The comparison is done using the text T of size $m = 681$. As it was defined in the formal definition, we were calculating the distance between intended position of the soloist in the score i and proposed position by the algorithm j, at the time t. The pseudocode for calculating the intended position, based on the information about the type of mistake which happened and its position, is given in the Algorithm 8. The distance is calculated between the list returned by the Algorithm 8 and the list of calculated positions, by explained approaches, for each character in the pattern P. As the input are given list of

positions where mistakes happened, type of operation and pattern P, which is already formed. The process of forming the pattern is explained in the Algorithms 3, 4, and 5.

Algorithm 8 Calculating intended positions i of the soloist for the score T

```
1: function CALCULATEINTENDEDPOSITIONS(operation, positionsOfMistakes, T, P,
   i = 0)
2:     indendedPositions ← empty list
3:     for t = 1; t ≤ t.size; t++ do
4:         if t == positionsOfMistakes[0] and operation == insertion then
5:             do not increase i                    ▷ positionsOfMistakes are sorted
6:         else if t == positionsOfMistakes[0] and operation == deletion then
7:             i ← i + 2                 ▷ skip position where the mistake happened
8:         else
9:             i ← i + 1 ▷ normally increase counter for going to the next character in P
   or for the replacement operation
10:        end if
11:        remove first element of positionsOfMistakes
12:        add i to the list indendedPositions
13:    end for
14:    return indendedPositions
15: end function
```

6.1.1 Insertion

The statistics for the number of wrongly suggested notes will be presented. Chapter 5 explained how every algorithm performs on string of small size. After running this algorithm on a much bigger string we compared three algorithms: naïve approach, reward based with and without filter. Number of errors for each iteration are presented in the Figure 6.1 for the number of errors $k = 3$ and in the Figure 6.2 for the number of errors $k = 10$. X axis represents the number of iteration and y axis the number of wrongly suggested positions by the algorithm.

While observing results, it was noticed that the algorithm returns a big number of wrongly suggested notes for cases of repetitive structure of the

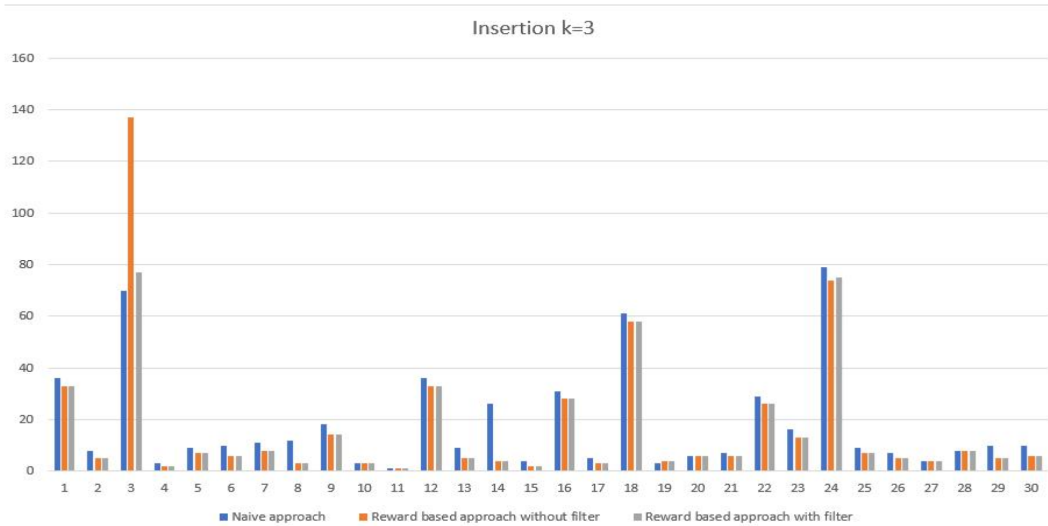

Figure 6.1: Results of random insertion into the string of size $m = 681$ and number of errors $k = 3$.

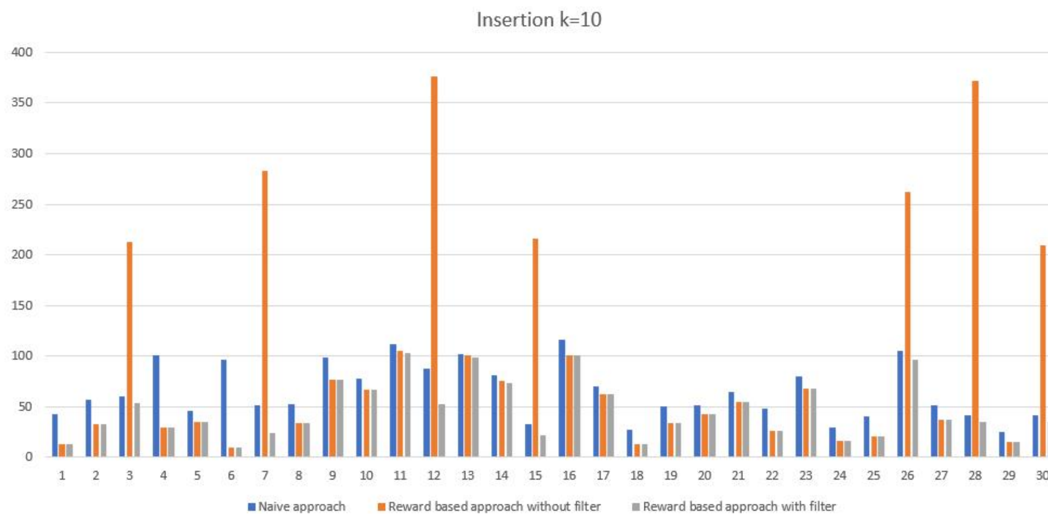

Figure 6.2: Results of random insertion into the string of size $m = 681$ and number of errors $k = 10$.

score. The reward based algorithm without filter performs badly with insertion errors which positions are relatively close to each other.

From the Figures 6.3 and 6.4 it can be noticed that the reward based algorithm with the filter performs the best. The reward based algorithm without the filter performs the worst and has the biggest dispersion. It was also noticed that the difference between intended position i and proposed position j in the score by the reward based algorithms was smaller than the one proposed by the naïve approach. Comparing to the other two approaches, it can be concluded that the best solution is the reward based solution with Kalman filter. Unlike the other algorithms, its number of wrongly proposed positions does not increase drastically by the growth of the number of insertions in the pattern P.

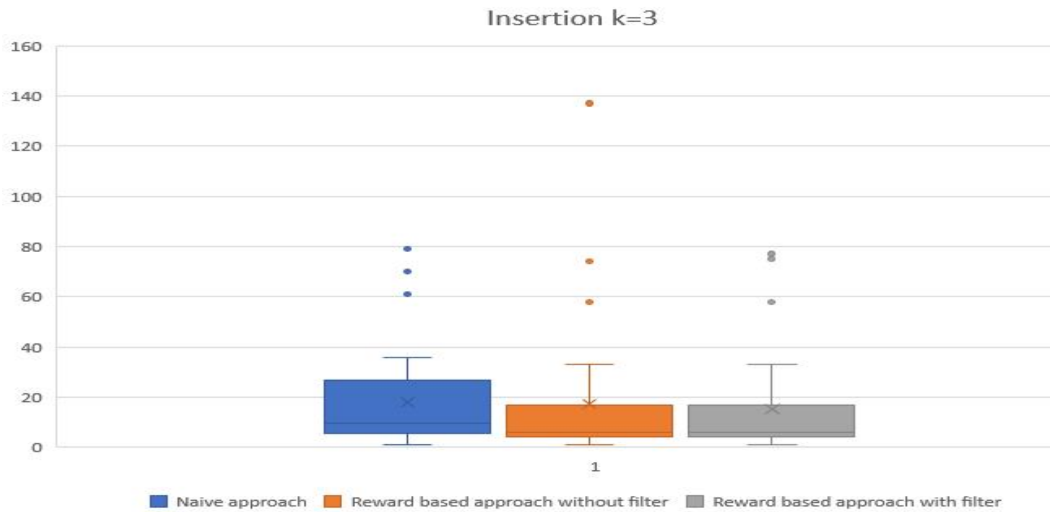

Figure 6.3: Box and Whisker plot for the string of size $m = 681$ and $k = 3$.

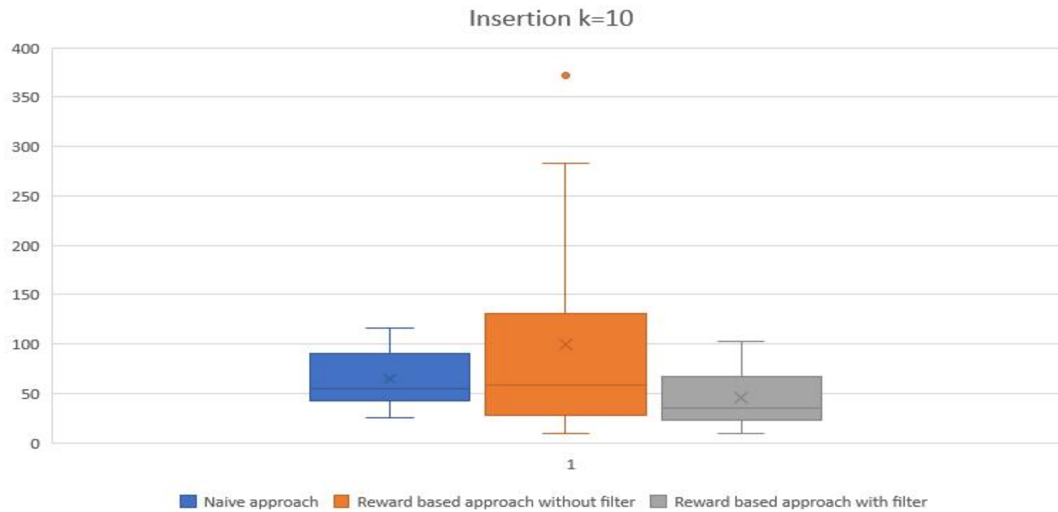

Figure 6.4: Box and Whisker plot for the string of size $m = 681$ and $k = 10$.

6.1.2 Replacement

When testing the replacement, it was noticed that the reward based algorithm gives better results than the naïve approach. The only time when the reward based algorithm without the filter had problems is when errors were on the positions relatively close to each other. It has been shown that this was not a big problem for replacement and that the algorithm recovers fast from it. Given the same string of size $m = 681$ statistics for the number of errors $k = 3$ and $k = 10$ are shown in the Figures 6.5 and 6.6.

Figure 6.7 and 6.8 represent the Box and Whisker plot for these results. Number of mistakes for the reward based algorithm without the filter is mostly 0. The cases when it is bigger than zero at the ones when two replacements happens relatively close to each other. As it can be seen from the plot this does not represent a big problem.

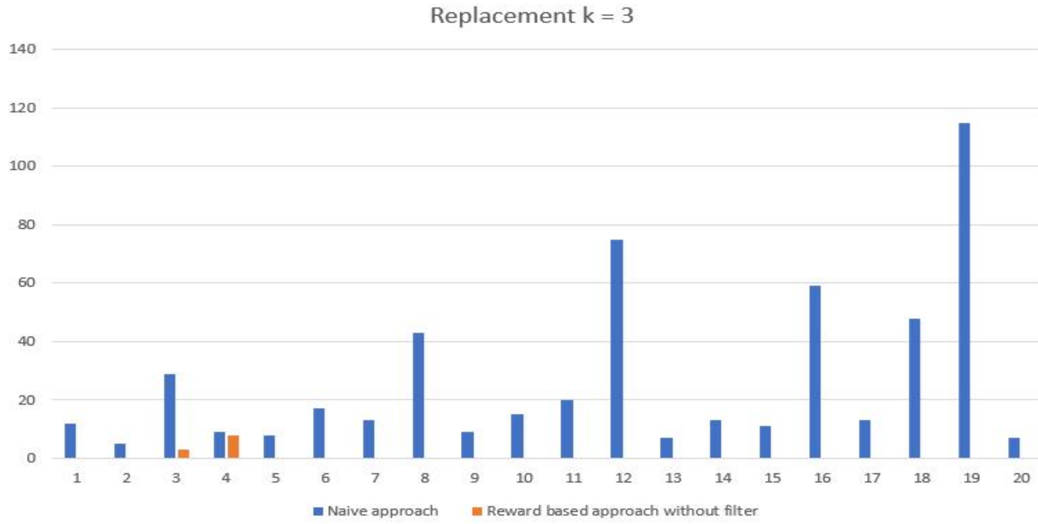

Figure 6.5: Results of random replacement for the string of size $m = 681$ and number of errors $k = 3$.

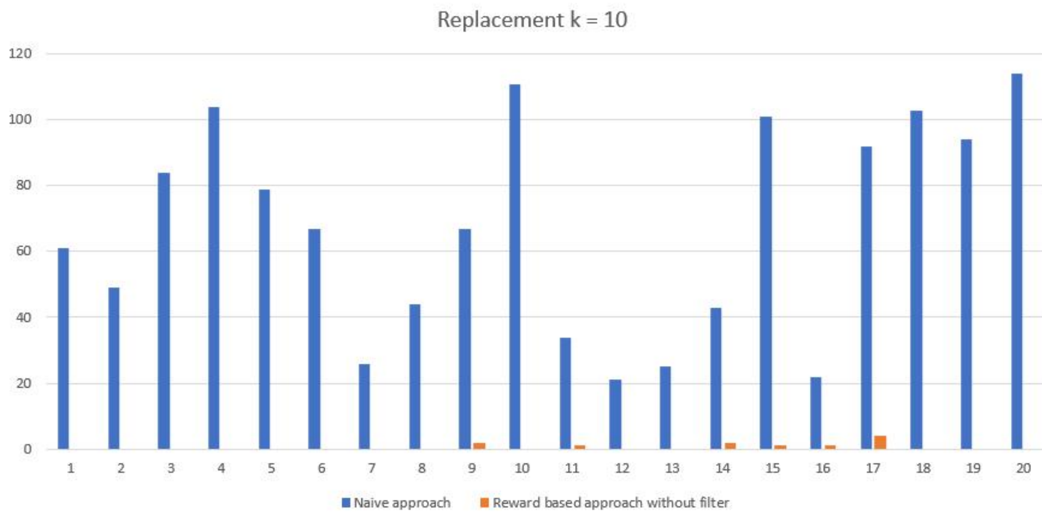

Figure 6.6: Results of random replacement for the string of size $m = 681$ and number of errors $k = 10$.

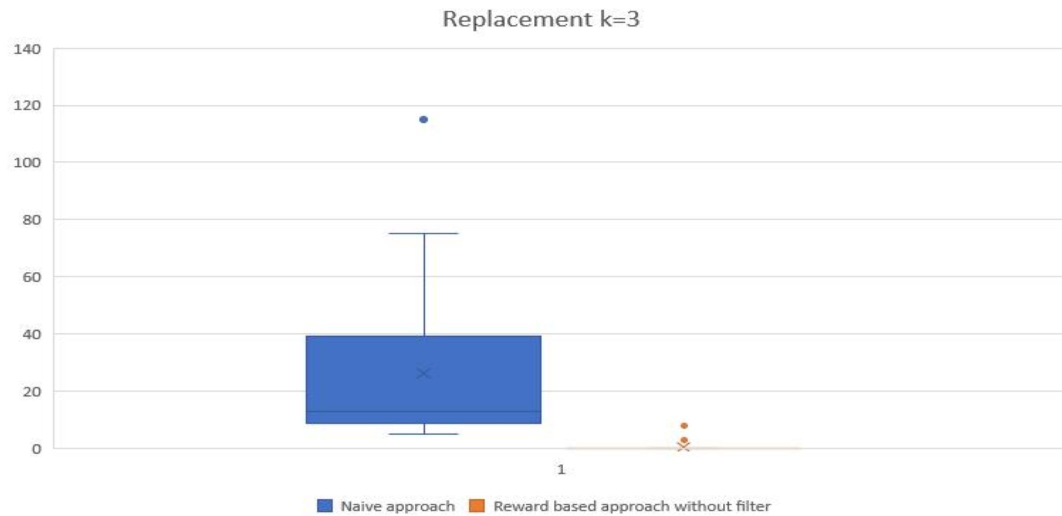

Figure 6.7: Box and Whisker plot for the string of size $m = 681$ and $k = 3$.

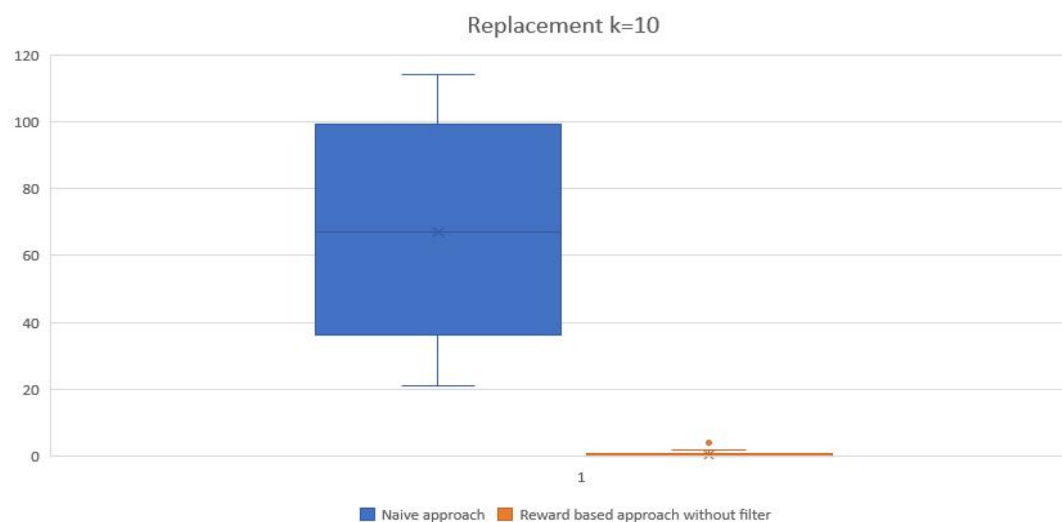

Figure 6.8: Box and Whisker plot for the string of size $m = 681$ and $k = 10$.

6.1.3 Deletion

The results for the same string of size $m = 681$ and using the pattern P of $k = 3$ and $k = 10$ random deletions are shown in the Figures 6.9 and 6.10 respectively.

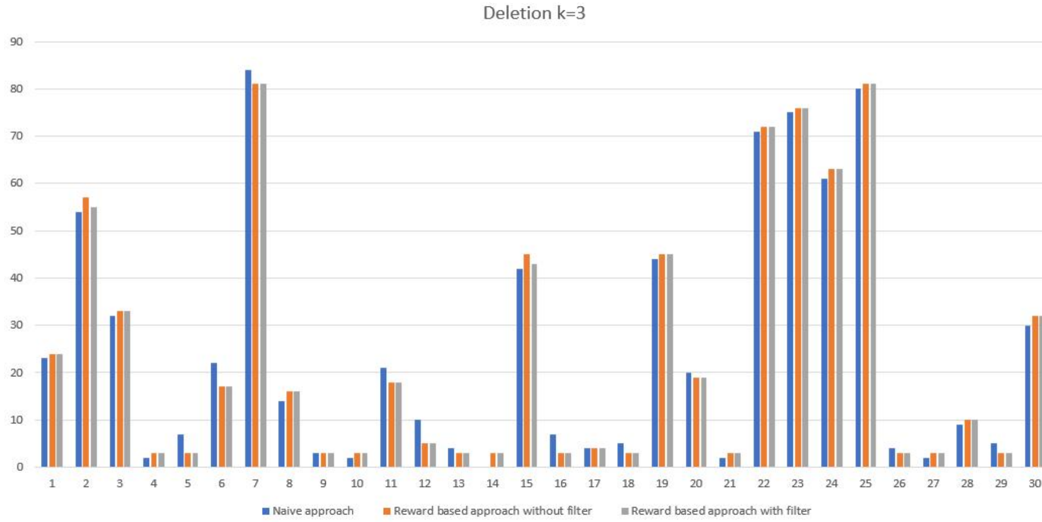

Figure 6.9: Results of random deletion for the string of size $m = 681$ and number of errors $k = 3$.

Given the plot from the Figures 6.11 and 6.12 it can be concluded that all three algorithms perform similar. The reward based approach without filter gave the worst results since the problem of deletions at the positions relatively close to each other can lead up to 500 mistakes for one iteration (see iteration 22 at the Figure 6.10). On the other hand, the version with the filter gave the best results. Furthermore, it was noticed that even with the bigger number of deletions, in the pattern P, algorithms perform again similar. The best is still reward based algorithm with Kalman filter because of the smallest dispersion with bigger number of mistakes.

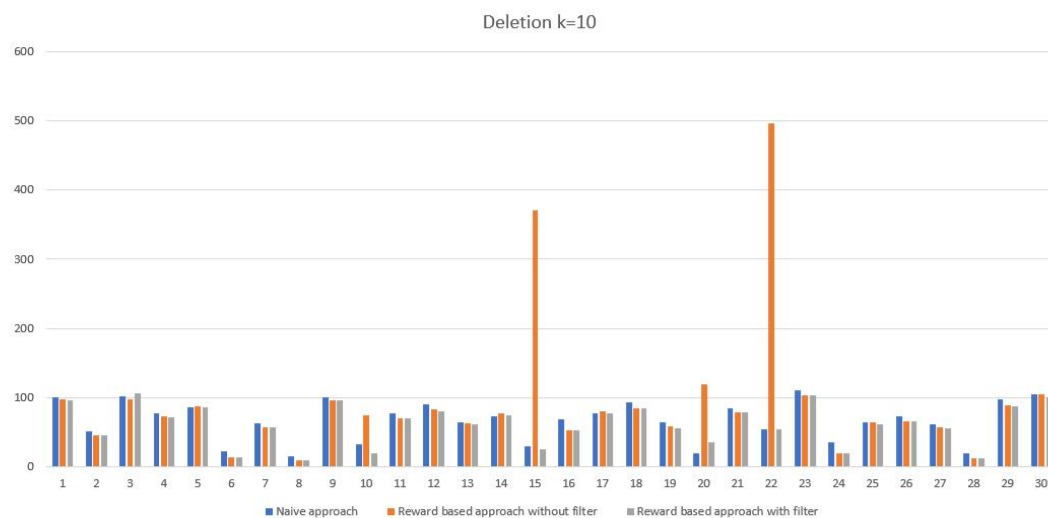

Figure 6.10: Results of random deletion for the string of size $m = 681$ and number of errors $k = 10$.

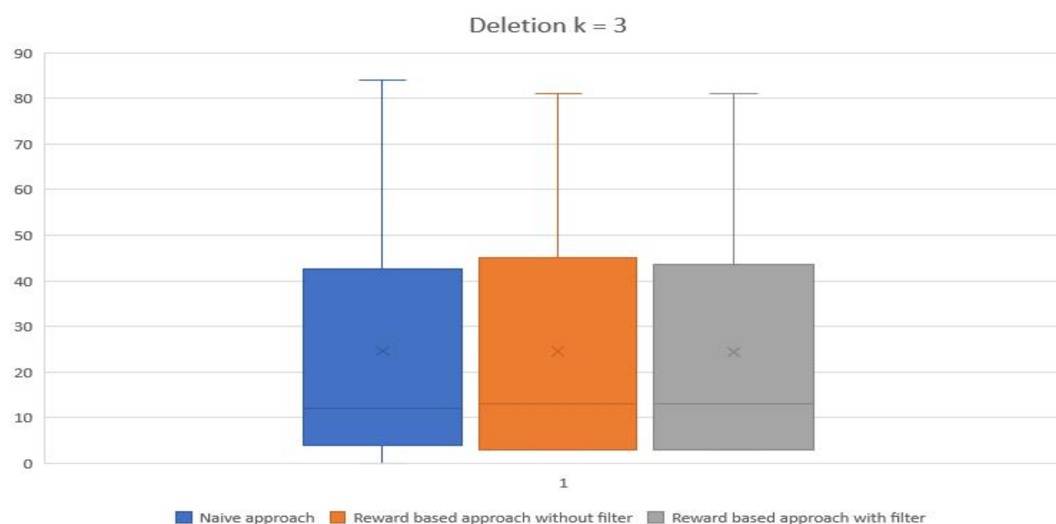

Figure 6.11: Box and Whisker plot for the string of size $m = 681$ and $k = 3$.

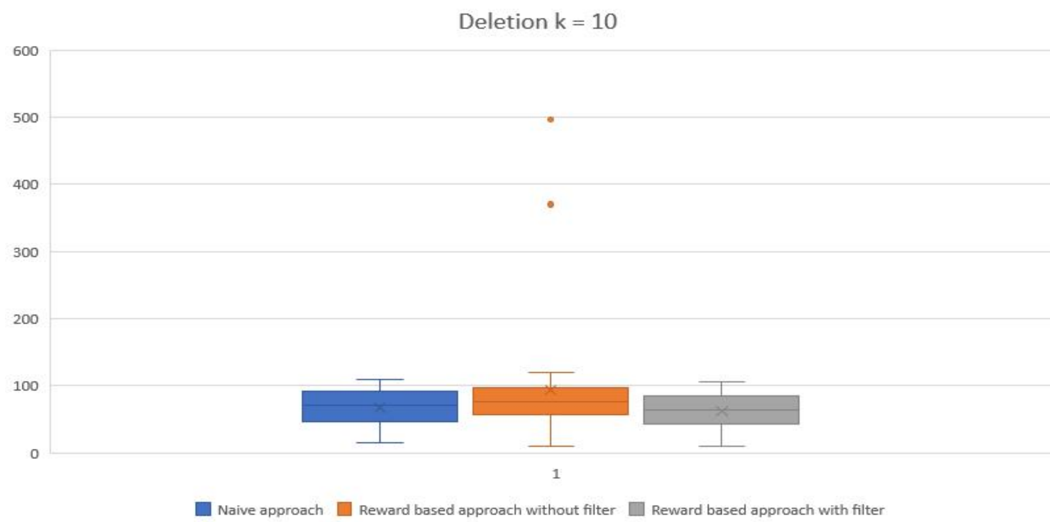

Figure 6.12: Box and Whisker plot for the string of size $m = 681$ and $k = 10$.

6.1.4 Evaluation for the jumps in the score

Another problem that we are solving with the play-along programme is to enable the soloist to make jumps in the score. We are approaching this problem by using the suffix tree data structure. We tested mentioned three algorithms for the same string T of size $m = 681$ and the number of errors $k = 3$. The results for each operation of insertion, replacement and deletion are shown in the Figures 6.13, 6.14 and 6.15 respecitvely. The patterns for testing are generated randomly. Primarily, two numbers $100 < rand1 < 300$, and $400 < rand2 < 681$ are randomly generated and substring between those two numbers it taken out of the pattern. This means that the pattern is $P = T[0..rand1, rand2..m]$. After that, operations of insertion, replacement and deletion are performed on the pattern, as in the Algorithms 3, 4, and 5.

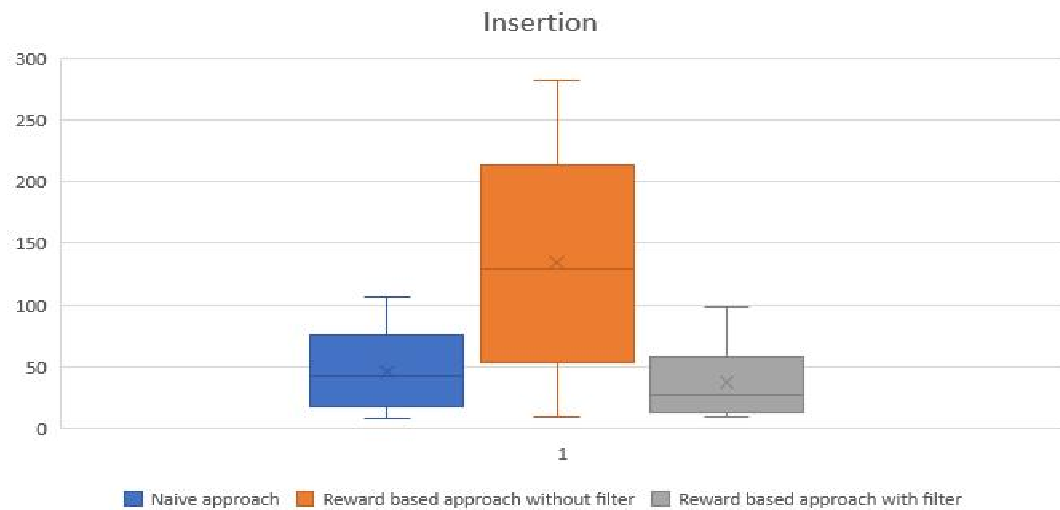

Figure 6.13: Box and Whisker plot for the pattern P with insertions and jumps in the score.

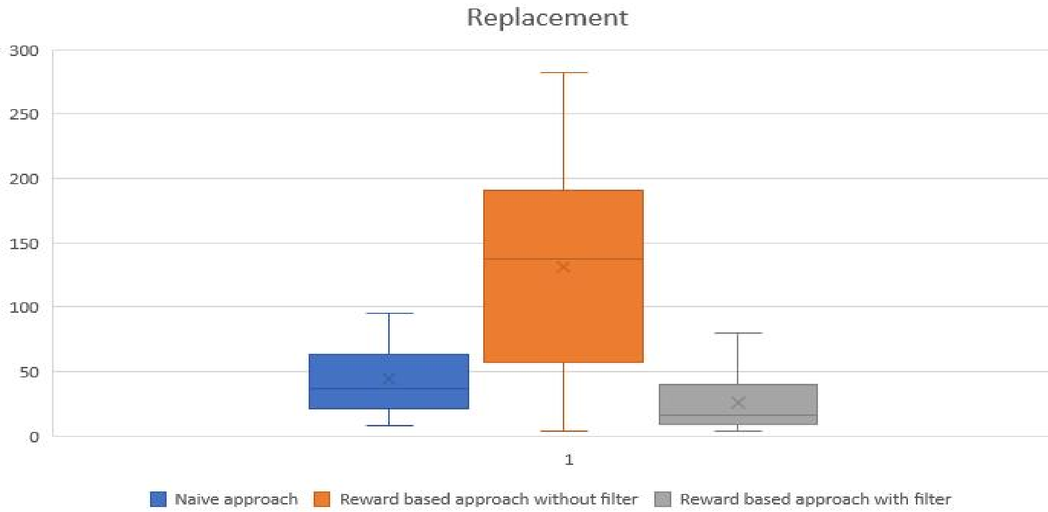

Figure 6.14: Box and Whisker plot for the pattern P with replacements and jumps in the score.

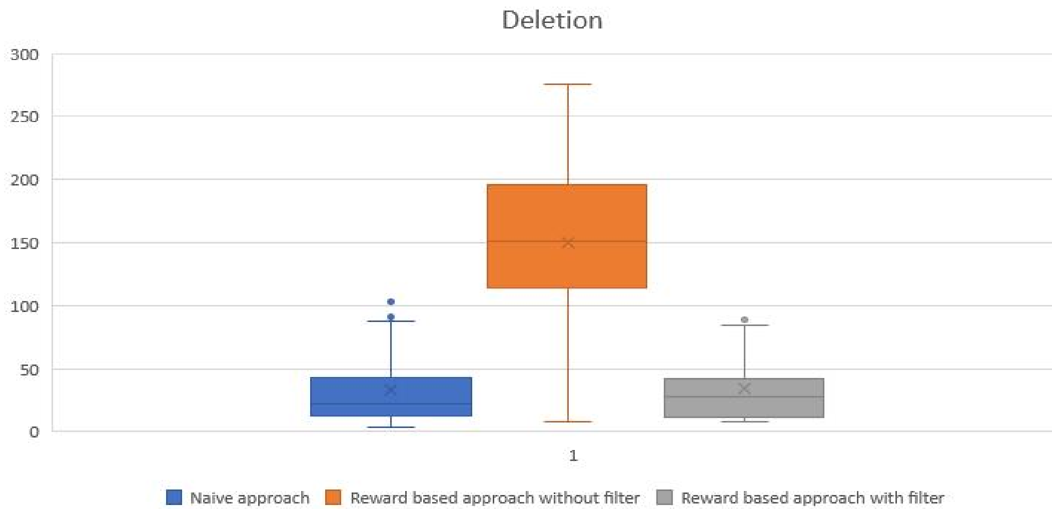

Figure 6.15: Box and Whisker plot for the pattern P with deletions and jumps in the score.

We can make the following observation. The reward based algorithm without the filter does not perform so well for replacement anymore. The

reason for this is that the size of text T is not as big as before. Furthermore, the naive solution is still better than the reward based approach without filter. Lastly, the reward based approach using the filter again performs the best. This happens because the reward based approach with Kalman filter handles mistakes most gracefully. The only problem all three algorithm have, and where most mistakes happen, is repetitive structure of the text T.

6.2 Complexity

Complexity of building a suffix tree is given with Ukkonen's algorithm, and it is $O(m)$. Complexity of walking down the tree for the pattern P of size n is $O(n)$. For the naïve approach walk down the tree for each character can be performed in constant time $O(1)$. Regarding the reward based approach, the algorithm for adding elements to the table is not constant time. If we denote the biggest outdegree of any node in τ as variable λ, and σ as the alphabet of the text T, then the algorithm time complexity is $O(\lambda \cdot \sigma)$.

Chapter 7

Conclusion

In this thesis, we proposed three types of solutions for the problem defined as an incremental approximate string matching problem. The problem of play-along is solved by finding the current position of the soloist in the score, and playing accompanist part for the next position. Also, the algorithm is taking into consideration that the soloist can make jumps in a performance. That is, the soloist does not have to play the score from the beginning until the end. (S)he can make jumps in the score and the algorithm should recover fast from it. The first solution was based on a simple walk down the suffix tree and returning to the root in the case of a mistake, while counting the number of the correct notes. The second solution was based on calculating rewards for the proposed positions, which will make the jumps harder for the notes played in the relatively close time frame. The approach had problems with intentional jumps of the player in the score, which was solved by introducing the third approach. This approach was also based on the rewards for suggested positions in the score, but it was filtering the old events using the Kalman filter. Implementation was done using C++. We tested three given approaches against each other for the edit distance operations: insertion, replacement, and deletion. The following observations are made:

1. Reward based algorithm with and without filter always give the same number of incorrectly proposed positions if mistakes in the pattern P

are not relatively close to each other.

2. If positions of mistakes are close, reward based algorithm without filter performs the worst, since it takes a lot of iterations for it to recover.

3. Reward based algorithms always propose positions which are very close to the intended ones, while the naïve approach has high jumps and bigger difference.

4. All three algorithms have problems with the repetitive structure of the score, the string T, as it is difficult for the algorithm to determine the right position in the text T of the character in the pattern P, given the nature of the suffix tree structure. Even with this structure, algorithms eventually find the right positions, just with the larger number of iterations.

5. Even when the soloist makes the jumps while playing the score, all three algorithms have no problems following.

6. The reward based approach with filter gave the best results, while the approach without the filter gave the worst results.

The code is published on GitHub repository [1], and it also contains methods for testing the proposed algorithms.

The emphasis of the implementation of all three approaches was on the regularity in the performance. There can be improvements in the algorithm complexity. Since in our case the text was relatively short, we did not encounter any problems during the evaluation. For a larger scores the algorithm would require a more efficient way of storing the node and other information in it.

[1] https://github.com/spica29/suffixTree

Bibliography

[1] E. Nakamura, P. Cuvillier, A. Cont, N. Ono, S. Sagayama, Autoregressive hidden semi-markov model of symbolic music performance for score following, in: 16th International Society for Music Information Retrieval Conference (ISMIR), 2015.

[2] A. Jordanous, A. Smaill, Investigating the role of score following in automatic musical accompaniment, Journal of New Music Research 38 (2) (2009) 197–209.

[3] R. B. Dannenberg, An on-line algorithm for real-time accompaniment, in: Proceedings of International Computer Music Conference, Vol. 84, 1984, pp. 193–198.

[4] B. Vercoe, The synthetic performer in the context of live performance, in: Proceedings of International Computer Music Conference, 1984, pp. 199–200.

[5] B. Vercoe, M. Puckette, Synthetic rehearsal: Training the synthetic performer, in: Proceedings of International Computer Music Conference, Vol. 85, 1985, pp. 275–278.

[6] M. Puckette, C. Lippe, Score following in practice, in: Proceedings of the International Computer Music Conference, 1992, pp. 182–182.

[7] B. A. Pardo, W. Birmingham, Following a musical performance from a partially specified score, in: Proceedings of the 2001 Multimedia Technology and Applications Conference, 2001.

[8] N. Orio, S. Lemouton, D. Schwarz, Score following: State of the art and new developments, in: Proceedings of the 2003 conference on New interfaces for musical expression, National University of Singapore, 2003, pp. 36–41.

[9] A. Cont, D. Schwarz, N. Schnell, Training ircam's score follower [audio to musical score alignment system], in: Acoustics, Speech, and Signal Processing, 2005. Proceedings.(ICASSP'05). IEEE International Conference on, Vol. 3, IEEE, 2005, pp. iii–253.

[10] N. Orio, D. Schwarz, Alignment of monophonic and polyphonic music to a score, in: Proceedings of International Computer Music Conference, 2001, pp. 1–1.

[11] J. J. Carabias-Orti, F. J. Rodríguez-Serrano, P. Vera-Candeas, N. Ruiz-Reyes, F. J. Cañadas-Quesada, An audio to score alignment framework using spectral factorization and dynamic time warping., in: 16th International Society for Music Information Retrieval Conference (ISMIR), 2015, pp. 742–748.

[12] D. Gusfield, Algorithms on strings, trees and sequences: computer science and computational biology, Cambridge university press, 1997.

[13] G. Navarro, A guided tour to approximate string matching, ACM computing surveys (CSUR) 33 (1) (2001) 31–88.

[14] V. I. Levenshtein, Binary Codes Capable of Correcting Deletions, Insertions and Reversals, Soviet Physics Doklady 10 (1966) 707.

[15] J. B. Kruskal, D. Sankoff, Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, Addison-Wesley, 1983.

[16] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, J. Kärkkäinen, Episode matching, in: Annual Symposium on Combinatorial Pattern Matching, Springer, 1997, pp. 12–27.

[17] S. B. Needleman, C. D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, Journal of molecular biology 48 (3) (1970) 443–453.

[18] P. Weiner, Linear pattern matching algorithms, in: Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on, IEEE, 1973, pp. 1–11.

[19] E. M. McCreight, A space-economical suffix tree construction algorithm, Journal of the ACM (JACM) 23 (2) (1976) 262–272.

[20] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.

[21] A. Rao, F. Lau, Automatic music accompanist, arXiv preprint arXiv:1803.09033 (2018) 1 – 5.

[22] R. E. Kalman, A new approach to linear filtering and prediction problems, Journal of basic Engineering 82 (1) (1960) 35–45.