Improving Automatic Task Tree Generation With Alignment Algorithms

Ralph Krimmel

am Institut für Informatik

Bachelor- und Masterarbeiten des Zentrums für Informatik an der Georg-August-Universität Göttingen

30. September 2014
Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 30. September 2014
## Contents

1 Introduction .......................................................... 4

2 Foundations .......................................................... 5
   2.1 GUI-Model .................................................. 5
   2.2 Task Tree Terminology .................................. 6
   2.3 State Of The Art Of Trace Based Task Tree Generation .................................................. 10
   2.4 Alignments .................................................. 12
   2.5 Substitution Matrixes ..................................... 12

3 Approach ............................................................ 14
   3.1 Task Tree Generation ..................................... 14
   3.2 Task Distance Substitution Matrix ..................... 14
   3.3 Iteration Detection And Substitution Matrix Update .................................................. 19
   3.4 Sequence Detection ...................................... 19

4 Implementation ..................................................... 28
   4.1 Substitution Matrix ....................................... 28
   4.2 Alignments .................................................. 29
   4.3 Match Search, Sorting and Replacement .............. 29

5 Case Study ............................................................ 31
   5.1 Parameters .................................................. 33
   5.2 Data Preprocessing ....................................... 33
   5.3 Calculation Of Distances Between Non-Event-Tasks .................................................. 34
   5.4 Evaluation Of Termination Conditions ................. 37
   5.5 Performance Evaluation ................................... 38
   5.6 Generated Task Trees .................................... 40
   5.7 Discussion .................................................. 43

6 Conclusion ............................................................ 44

Bibliography ............................................................ 45
List of Figures

2.1 Excerpt from a GUI-Model of the website www.app.informatik.uni-goettingen.de

2.2 Simplified and not complete visit of an online book shop as a task tree

2.3 An example for a task tree with temporal relations. Adopted from Harms et al.[5]

2.4 An example for one task tree instance of the task tree in figure 2.3. Adopted from Harms et al.[5]

2.6 Example for one repetition of the iteration and sequence detection for task tree generation. Figure from Harms et al.[5]

2.7 Three possible modifications of sequence

2.8 An example for substitution matrixes: the PAM250 substitution matrix for amino acids[3]. Note that biological substitution matrixes may not be symmetric

3.1 Web form with multiple text input fields with large absolute distance inbetween

3.2 A HTML GUI-model with two nodes (green) the distance will be calculated for. The number of red nodes is the distance between the green nodes. The common anchor is the <body> element.

3.3 Illustration of the three operations deletion, insertion and substitution in a dynamic programming matrix

3.4 Figure from Durbin et al.[4]. The repeat dynamic programming matrix for two example sequences, for T = 20. Below the optimal alignment, with total score 9 = 29 − 20. There are two separate match regions, with scores 1 and 8. Dots are used to indicate unmatched regions of x. The arrows show the traceback process, going back from the total score to the start.

3.5 Example for a retrieved match. Each of the two subsequences of the sequences the match was generated from is printed in a row.

3.6 Example for rule 1. Will only match a user session subsequence 16 4.

3.7 Example for rule 2. Will match user session subsequences 16 4 6 and 16 6.

3.8 Examples for rule 3 (a) and (b).

3.9 Example for rule 4. Match is found if a subsequence of a user session is 15,17,16 or 16,18,14 but not 15,18,14.
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.10</td>
<td>Example for task generation. The match on the left is converted into the task on the right.</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>Screenshot of the website the data was collected with.</td>
<td>31</td>
</tr>
<tr>
<td>5.2</td>
<td>Example of one event in XML structure.</td>
<td>32</td>
</tr>
<tr>
<td>5.3</td>
<td>An example for a task found by alignment task tree generation with distance calculation between non-event-tasks.</td>
<td>35</td>
</tr>
<tr>
<td>5.4</td>
<td>Graph with matches found per repetition and the time per repetition.</td>
<td>38</td>
</tr>
<tr>
<td>5.5</td>
<td>Scaling behaviour of both approaches.</td>
<td>39</td>
</tr>
<tr>
<td>5.6</td>
<td>Example for a task tree consisting of two different interactions made by the user.</td>
<td>41</td>
</tr>
<tr>
<td>5.7</td>
<td>Example for a created task that shows that further data preprocessing is required.</td>
<td>42</td>
</tr>
<tr>
<td>5.8</td>
<td>Example for a task for setting a new password.</td>
<td>42</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example for a trace of the login process of a user [5].</td>
<td>6</td>
</tr>
<tr>
<td>5.1</td>
<td>Case study overview.</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>Table of all parameters of the alignment approach that can be adjusted for</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>the task tree generation</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Results of the version with and without calculation of the distances between</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>non-event-tasks</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Matches found per repetition and the time for each repetition. All times in</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>minutes</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Comparison between execution times (seconds) of n-gram approach and</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>alignment supported task tree generation. Note that the execution time of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>the alignment approach for 10 user sessions seems to be an approximation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>of πt.</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>Comparison of the n-gram approach from Harms et al. and the alignment</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>approach. The alignment approach finds less task but is able to detect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>selections and optionals.</td>
<td></td>
</tr>
</tbody>
</table>
1 Introduction

Usability has become a very important aspect of software development. Once an interface is designed badly users tend to use the software inefficiently or stop using it at all due to frustrating experiences. Therefore, it is critical to test the usability during software design but also evaluate the feedback given from users. This feedback can be gathered automatically by observing how users actually use the software. In this work we will improve existing methods for evaluating collected user interactions.

Task trees are one well method to describe and structure the process of users interacting with software. We define them as a type of task model which describes user actions. There are two possibilities to create a task tree. First of all, the designer of the software can generate a task tree at design time to model the expected user behaviour[5]. This approach is used by ConcurTaskTrees[9], a notation for task model specification and ReverseAllUI[1]. With those methods it is possible to extract task models from several markup languages like HTML, XHTML or TERESA XML. The second option is to generate the task trees based on observed user interactions. Those task trees represent the effective user behaviour.

There are several approaches to generate task models automatically from user traces e.g. CRITIQUE[7], which creates GOMS (Goals, Operators, Methods and Selection rules) models. Harms et al. described a method to create task trees from observed user interactions[5]. A comparison of the effective and the expected user behaviour offers the possibility to semi-automatically evaluate the usability of the software.

In this work we try to improve the approach from Harms et al. by applying methods usually common in bioinformatics. We will first cover the foundations needed in the following chapters and then describe our approach in chapter 3. After that, we will give some details about the implementation of our approach. In chapter 5 we will evaluate our method by applying it on real data and discuss our results.
2 Foundations

In this chapter we will provide and explain the concepts and terms that are used throughout this thesis. We will cover the definitions of a GUI-Model as well as the basic terminology of a task tree, followed by information on alignment algorithms and substitution matrixes. In the last section we describe the state of the art task tree generation based on n-grams.[5]

2.1 GUI-Model

A GUI-model is the structure of a graphical user interface. It consists of different elements which themselves can have elements as children. A GUI-model can be displayed as a tree. Figure 2.1 shows a part of the GUI-model of the case study we will use in chapter[5]

Server(www.app.informatik.uni-goettingen.de:80)


html[0]

body[0]

group_[itlis/cs2014ss]/[2014/accounts|application|wizard|imprint]/[wise|login|password]

div[0]

group_/cs2014ss/application

p[0]

group_/itlis/cs2014ss/[2014/wizard]/[wise|degree|en-test]/wizard/[degree|en-test]

div(id="date-chooser")

div[2]

div[1]

div(id="date-chooser_day")

Figure 2.1: Excerpt from a GUI-Model of the website www.app.informatik.uni-goettingen.de.
2 Foundations

2.2 Task Tree Terminology

Harms et al. [5] have created a set of definitions and the respective vocabulary that help dealing with task trees. We will adopt their definitions and summarize them in this section.

To understand tasks and task trees we have to look at how users interact with software. Users, performing actions like clicking or typing, trigger so called events. Those events consist of a type and a target. The type describes the type of the action that triggered this event (e.g. onclick), the target describes the element in which the GUI the event occurred on. The events a user creates can be traced, which is the basic step to gather data that is needed to generate task trees. A trace is a list of events in the order they occurred. Table 2.1 shows an example of such a trace.

All events in this list and their corresponding actions represent the users intention to achieve a goal, while using the software e.g. buying a book in an online shop. From now on this goal is named task. This task can consist of several subtasks such as logging in or putting a book into the basket. Each subtask itself can again contain further subtasks. In figure 2.2 we can see this results in a tree structure, a so called task tree. A task tree consists of three different kinds of nodes, the root node, the leaf nodes and the intermediate nodes.

<table>
<thead>
<tr>
<th>Order</th>
<th>Type</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Left mouse button click</td>
<td>Textfield with ID username</td>
</tr>
<tr>
<td>2</td>
<td>Text input &quot;usr&quot;</td>
<td>Textfield with ID username</td>
</tr>
<tr>
<td>3</td>
<td>Left mouse button click</td>
<td>Textfield with ID username</td>
</tr>
<tr>
<td>4</td>
<td>Text input &quot;user&quot;</td>
<td>Textfield with ID username</td>
</tr>
<tr>
<td>5</td>
<td>Left mouse button click</td>
<td>Textfield with ID password</td>
</tr>
<tr>
<td>6</td>
<td>Text input &quot;&quot;</td>
<td>Textfield with ID password</td>
</tr>
<tr>
<td>7</td>
<td>Left mouse button click</td>
<td>Button with name &quot;login&quot;</td>
</tr>
</tbody>
</table>

Table 2.1: Example for a trace of the login process of a user [5].

Root node

Represents the overall task which contains all subtasks, the user wants to achieve this goal by his actions/his input. In figure 2.2 the overall task is to buy books at an online shop.

Leaf nodes

Actions (e.g. clicking, scrolling, textinput) cause an event (e.g. onclick (click) or on-change (textinput)). The tasks at this level are called event-tasks. Event-tasks do not have any further children. Some event-tasks of 2.2 are for example Input A, Input B and Input C.
Figure 2.2: Simplified and not complete visit of an online book shop as a task tree.
2 Foundations

Intermediate nodes

Subtasks which are steps towards the overall task. All tasks above the leave nodes are called non-event-tasks. In figure 2.2, the intermediate nodes are login, search, basket and order.

Another relevant point is that the order of the executed actions is important. Therefore, this information has to be represented in the task tree as well. For this, Harms et al. defined the so called temporal relationships for all non-event-tasks. There are four types of temporal relationships, each with different properties. The sequence temporal relationship dictates that children have to be executed in specific order to fulfil the task. An iteration can just have one child, which is executed once or several times. Just one child of a selection is allowed to be executed, whereas the one child of an optional may or may not be executed. Figure 2.3 shows a task tree that represents a login procedure and contains all possible temporal relationships. At last, Harms et al. differentiate between a task and an executed task, a task instance. Figure 2.4 shows the main differences between a task and its instance: selection instances can have only one child with the task instance that was executed. Optional instances have one child if the task was executed and none if the task was not executed. Instances of iterations have as many children as often as the task was executed.

Figure 2.3: An example for a task tree with temporal relations. Adopted from Harms et al.[5].
Figure 2.4: An example for one task tree instance of the task tree in figure 2.3 Adopted from Harms et al. [5].
2.3 State Of The Art Of Trace Based Task Tree Generation

The current procedure to generate task trees\[5\] starts with iterating through the events in the traces and creates an event-task instance for each. Those event-task instances are stored in the order they were triggered. This structure is a so called user session.

The user sessions used for task tree generation are available in a non-harmonized form, meaning equal task instances may not have the same corresponding task assigned. The now following harmonization process sets the correct task to those task instances. Fixing the issue of non-harmonized input data before starting any further steps is crucial since it reduces the number of occurring tasks and makes task instances comparable by their tasks. Figure 2.5a shows a non-harmonized user session, figure 2.5b a session where the tasks of equal task instances have been set accordingly.

After the harmonization step follows the iteration detection, which detects any identical task instances that are directly adjacent task instances in the user sessions. They occur if a user repeatedly does the same action (e.g. multiple clicks on the same button). Once those subsequent instances have been detected, a new task of type iteration is generated. We recall from the previous section that a task with a temporal relationship iteration can just have one child, so the task of the detected iterated instance is added to this newly created iteration task. Afterwards, this task is replaced in every other place it occurs as well. In contradiction to the task, the iteration instance has, after the replacement, as many children of the instance as often the iterated instance occurred.

The next part of the task tree generation algorithm is the sequence detection. For this, the user sessions are scanned for identical subsequences. This is done with the help of a trie data structure. The longest subsequence that occurred most often is then added to a newly created sequence task and again replaced in all other user sessions as well. After that, the next most occurring subsequence is replaced. The minimal length of a subsequence is two. If there is more than one subsequence with equal occurrence count, just the longest subsequence is replaced. If both occurrences match in count and length, Harms et al. just replace the subsequence occurring first in the ordered list. Both steps, iteration detection and sequence detection are repeated until no more iterations or sequences are found. Figure 2.6 shows an example of the first repetition of the task tree generation.

<table>
<thead>
<tr>
<th>Task</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task instance</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

(a) Non-harmonized user session.

<table>
<thead>
<tr>
<th>Task</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task instance</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

(b) Harmonized user session. Task instance A now always has 1 as its task, task instance B always 2.
Figure 2.6: Example for one repetition of the iteration and sequence detection for task tree generation. Figure from Harms et al.
2.4 Alignments

Alignments are a method in bioinformatics to compare strings or sequences of DNA, RNA and amino acids and score their similarity. It is a very basic method to find out if two sequences are related. This is done by arranging those sequences in a way that similar elements of each sequence are aligned together. The biological background of this approach is that both sequences have diverged from a common ancestor by a process of mutation and selection\[^4\], hence still have common regions. Those regions are called conserved regions, meaning they were resistant to mutation.

In contrast to the persistent regions, differences between sequences can be explained by three processes: substitution, which replaces one element with another, deletion, which removes an element and insertion, which adds a new element to a sequence. Both, insertion and deletion, produce a so called gap in the alignment, since the change just appeared in one of the two aligned sequences. Figure 2.7 illustrates all possible modifications. Each modification can be scored individually but it is common practise to score substitutions with substitution matrixes and have a general gap penalty. The sum of all single scores is the score of the total alignment. Alignment algorithms try to maximize this score.

Alignments can be categorized into global and local alignments. Global alignment algorithms try to match one sequence with the other from start to end, whereas local alignment algorithms try to find the best alignment of subsequences. A popular global alignment algorithm is the Needleman-Wunsch algorithm\[^8\]. For basic local alignments the common algorithm is from Smith and Waterman\[^10\].

2.5 Substitution Matrixes

In the last section we stated that alignment algorithms maximize the total alignment score. Therefore, the underlying scoring model has to be adjusted very carefully, since it practically determines the outcome of the algorithm. There are usually two parameters which have to be set: the substitution score and the gap penalty.

**Definition 2.1**

Let $g$ be the gap penalty with $g > 0$. 

---

1. A B C D
2. A B C A

(a) Substitution of D

1. A B C D
2. A B - D

(b) Deletion of C

1. A B C -
2. A B C D

(c) Insertion of D

Figure 2.7: Three possible modifications of sequence 2.
The gap penalty in its most basic version is a constant that has to be manually adjusted so that it fits the problem well. A large gap penalty leads to less insertion and deletion occurrences and more substitutions. This type of gap scoring is called linear affine gap penalty. The total gap penalty is \( n \times g \) with \( n \) being the number of gaps and \( g \) the gap penalty constant. There are different gap penalty models like a different \( g \) for opening a new gap and extending an already open gap. In this thesis we will just use the linear affine gap penalty.

The substitution score is a representation of how similar two elements are. The more similar they are, the higher the score will be. The score is usually stored in symmetric matrices, where each cell represents how good or bad it is to substitute an element \( a \) with an element \( b \). The formal definition of the substitution matrix will follow in chapter approach.

---

**Figure 2.8:** An example for substitution matrixes: the PAM250 substitution matrix for amino acids. Note that biological substitution matrixes may not be symmetric.
3 Approach

The algorithm proposed by Harms et al. is well designed but is not capable of finding similar user sequences. When there is more than one possible interaction to achieve a goal, the method of Harms et al. will create two different sequences for that interaction, or worse, will not detect the interaction as a meaningful one at all. For this reason, we propose an algorithm that is able to detect similar subsequences. The basic steps of the algorithm do not differ a lot from Harms et al. one. In fact, some preprocessing steps and the sequence detection have been altered. Algorithm 1 shows the main building blocks and will function as a list and order of contents of this chapter.

3.1 Task Tree Generation

Algorithm 1 Overview over the task tree generation

```
procedure GENERATE_TASKTREE(UserSessions)
    Generate Substitution Matrix (UniqueTasks)
    while Replaced Tasks do
        Detect Iterations (UserSessions)
        Optional: Substitution Matrix Update
        Detect Sequences (UserSessions,SubstitutionMatrix)
    end while
end procedure
```

3.2 Task Distance Substitution Matrix

The harmonization process performed by Harms et al. is also applied to the user sessions in this approach. A useful side product of the harmonization is the set of unique tasks. This set is needed for this step of the algorithm, the generation of the substitution matrix.

In section 2.5 we introduced substitution matrices in general and what they are used for. For the use case of detecting tasks we need to generate one substitution matrix that represents how similar tasks are. The score of two tasks in the task distance substitution matrix is defined as follows:
Definition 3.1
Let $a$ and $b$ be two tasks.
Let $S(a,b) = S(b,a)$ be the score for substituting task $a$ with task $b$. The higher the value of $S$ is, the more similar are the tasks $a$ and $b$ and vice versa.

To calculate the score, three cases have to be considered:

- Similarity of two event-tasks
- Similarity of an event-task and a non-event-task
- Similarity of two non-event-tasks

3.2.1 Event-Task To Event-Task Similarity

The first idea was to calculate the score of the matrix based on the distance between the absolute coordinates of the event-tasks. There are a few problems with this approach: first, not all event-tasks may have absolute coordinates. The second problem with this method is that in graphical user interfaces events may still be very similar, even if they have a large absolute distance. An example for such a case would be a web formular with many fields to fill out. Figure 3.1 shows such an example. Those fields take space which would result in large distances between them although the event-tasks all belong to a single formular. The solution to this problems is to make use of a grouping of elements the designer of the GUI already did: the GUI-Model (see section 2.1). Elements of a GUI that belong to one semantic task can usually be found in some kind of container that groups those elements together. Therefore, the basis for our distance calculation is the distance in the GUI-Model, as defined in 3.2.

Definition 3.2
Let $a$ and $b$ be event-tasks

\[ d(a, b) = d(b, a) = \text{the distance in the GUI-model of the targets of event-tasks } a \text{ and } b. \]

Since the GUI-model is a tree, the distance of two targets in a GUI can easily be calculated by finding the common ancestor of the targets and summing up the number of nodes from both the events to this ancestor, including the ancestor. Figure 3.2 shows a GUI-model with two elements and their common ancestor.
3.2.2 Event-Task To Non-Event-Task Similarity

It takes a bit more effort to calculate the substitution score if one task is a non-event-task. The reason is that the non-event-tasks do not represent a simple event anymore. Therefore, they do not possess a target in the GUI. A possible solution is to recursively visit every child of the non-event-task, gather all event-tasks and then calculate the mean distance from each of those tasks to the event-task the distance shall be calculated to. Formally, definition 3.2 has to be modified so that it covers non-events as well:
Definition 3.3
Let \( c \) be a non-event-task.
Let \( E \) be the set containing all event-tasks that can be recursively found in \( c \).
With definition 3.2 it is possible to define \( d \) as

\[
d(a, c) = d(c, a) = \frac{\sum_{x \in E} d(a, x)}{|E|}
\]

3.2.3 Non-Event-Task To Non-Event-Task Similarity

With definition 3.3 it is simple to compute the distance for two tasks since all that is to do now is to repeat the procedure of finding all event-task children of one task, calculate the distances to the other task and use the mean distances as the total distance. The definition of \( d \) can be extended so it accepts two non-event-tasks:

Definition 3.4
Let \( c \) and \( d \) be non-event-tasks.
Let \( E \) be the set containing all event-tasks that can be recursively found in \( c \).
With definition 3.3 it is possible to define \( d \) as

\[
d(c,d) = d(d,c) = \frac{\sum_{x \in E} d(x,d)}{|E|}
\]
3.2.4 Score

Now that we defined the distance for three cases it is possible to compute the score $S$ of two tasks. To transform a distance into a score we multiply all distance values with $-1$ and add a constant so that some scores are positive and some have a negative value.

**Definition 3.5**

Let $U$ be the set of unique tasks occurring in the user sessions.
Let $k$ be a constant that defines the maximal score.
For each tuple $i, j \in U$

$$S(i, j) = -1 \cdot d(i, j) + k$$

The $k$ constant should be chosen dependent on the underlying GUI-model. A large $k$ may be chosen for very deeply, nested GUI-models whereas for flat GUI-models a smaller $k$ seems better. At last this parameter has to be evaluated and carefully adjusted to the given input data.

The maximal score is reached if the distance between two elements is zero. This happens if we compare two equal elements. A problem can occur when the score of a non-event-task to an event-task is equal to maximal score. This happens if a non-event-task has just one event-task as child and the distance of this child to the same event-task is calculated. Since it is preferred that the score of two equal event-tasks is always larger than the score of this event-task with a non-event-task we add the penalty term $L$ to the score equation.

**Definition 3.6**

Let $E$ be the set of all event-tasks.
Let $N$ be the set of all non-event-tasks.

$$L(i, j) = \begin{cases} 
0 & \text{if } i, j \in E \\
\text{constant} & \text{if } i \in N \lor j \in N 
\end{cases}$$

$$S(i, j) = -1 \cdot d(i, j) + k - L$$
3.3 Iteration Detection And Substitution Matrix Update

The iteration detection from Harms is also suitable for our algorithm and was not altered in any way. It reliably detects iterations and replaces them in the user sessions. Before the sequence detection can come into play, the substitution matrix should be updated since there were new iteration tasks created during iteration detection. Those new tasks are stored in a set that contains all newly created tasks. The update process differs just a little from the generation process with the only difference being that just the distances between the newly created tasks to the current set of unique tasks is as well as the distances between the newly created tasks are computed. After the matrix has been updated, the newly created tasks are merged with the set of unique tasks and then emptied.

The update process is a very expensive procedure. An alternative to this is to set each score between an event-task to a non-event-task and therefore all scores between non-event-tasks to zero. Both variants are evaluated in chapter 5.

3.4 Sequence Detection

As we figured out at the beginning of this chapter the sequence detection is the part of the algorithm that varies the most from Harms approach. It is itself separated in four steps:

1. The search for significant patterns
2. The match retrieval
3. The task generation
4. The sequence replacement

3.4.1 Search For Significant Patterns

The search for significant patterns is done with an alignment method we introduced in section 2.4. The bioinformatical problem of comparing and aligning DNA/RNA/Amino acid sequences is closely related to finding more or less equal subsequences in the user sessions. The conserved regions of biological sequences comply with those interactions that most of the users performed similarly. We can conclude that those interactions are meaningful in the sense of fulfilling one desired task. The goal of the alignment algorithm is now to detect the conserved regions and extract a model of the average user behaviour. Since alignment algorithms also find approximate and not only exact similarities, this step of the sequence detection is the main difference to Harms method.

The first step of the search for significant patterns is to align every user session with any other user session with the Smith-Waterman algorithm for repeated matches. For the
rest of this section the user session will be represented by a sequence of numbers. Each number in a sequence corresponds with the unique ID of the task at this position in the user sessions. We now describe the alignment algorithm in detail.

**Smith Waterman Algorithm For Repeated Matches**

The Smith-Waterman algorithm for repeated matches is a modified version of the original Smith-Waterman algorithm. The original Smith-Waterman finds the best local similarity between two sequences. Since we need all relevant similarities and not just the one with the best score, we define a threshold score $T$.

**Definition 3.7** Let $T$ be the threshold score for subalignments.

Once a local subalignment reaches $T$ it is considered as relevant.

We recall from the section 2.4 that alignment algorithms try to find the best possible score between two sequences using the underlying scoring model of substitution and gap scores. The most naïve approach is to test possible combinations both sequences could be aligned. As usual, this approach is not very feasible because there are $\binom{2n}{n} \approx 2^{2n}\sqrt{\pi n}$ possible global alignments of two sequences. Another way would be to use some kind of heuristic but we are interested in an exact, deterministic method. A common technique to solve this issue is dynamic programming. The main idea of dynamic programming is not to calculate every possible variant but to reuse the best solutions of smaller subproblems. This method is central to computational sequence analysis.

**Basic Smith-Waterman** The basic version of the Smith-Waterman algorithm makes use of dynamic programming by defining the dynamic programming matrix $F$ where each cell of the matrix stores the best score for aligning the elements up to the position the cell represents. For example the cell $(2|2)$ stores the score for the best alignment of first two elements of each sequence. Each cell is computed by getting the best score out of four choices. Prior to enlisting those, we need the following definitions:

**Definition 3.8**

Let $x,y$ be two sequences. Let $F$ be a matrix with $F(i,j)$ being the best score of aligning $x_1 \ldots x_i$ with $y_1 \ldots y_j$. 


Then we can compute $F$ by repeatedly applying the following equation:

$$F(i,j) = \max \begin{cases} 
0, & \text{Gap} \\
F(i-1,j-1) + S(x_i, y_j), & \text{Substitution} \\
F(i-1,j) - g, & \text{Gap} \\
F(i,j-1) - g. & \text{Gap}
\end{cases}$$

(3.1)

Figure 3.3 illustrates this formula. The diagonal choice means we align one element from each sequence with each other. Therefore, we have to look up the similarity of both elements in the substitution matrix. When the score from the upper cell minus the gap penalty is bigger than the diagonal choice a gap will appear at this position in sequence $y$. This is the same for when the option from the left cell is taken, just that the gap will be inserted in sequence $x$. The option 0 in the equation is to prevent alignment scores to become negative. If a match cannot be extended with its score being a positive value, it is better to start a new one later. As we can see, it is possible to fill the whole matrix by once the trivial scores, the first row and the first column, have been initialized.

**Repeated Matches** In the improved version of the algorithm a cell in $F$ has a slight different meaning and its computation is a bit more complex. The first difference is that this algorithm is asymmetric in the sense of aligning sequence $x$ to sequence $y$ produces a different outcome than aligning sequence $y$ to sequence $x$. Hence, we define $y$ as the pattern we want to search and $x$ as the sequence in which we search all subsequences of $y$ repeatedly. This results in a final alignment where $x$ has matched and unmatched regions.

The algorithm itself is partitioned into three parts:

- Initialization
3 Approach

- Recursion
- Traceback

**Initialization** The initialization step is simple:
Let \( m \) be the length of \( y \).

\[
F(0, j) = 0 \text{ for } j = 0 \ldots m
\]

This differs from the Smith-Waterman algorithm described by Durbin et al.[4] who just initialize \( F(0, 0) \) and do not fill the first column. There is no need to initialize the first row because a subalignment will never start with a gap. For the sake of an easy implementation it was better to initialize the first row though.

**Recursion** The recursion formula for filling the matrix also has the three options for insertion, deletion and substitution as illustrated in figure [3.3]. But as we see there are some more options. For the first row (equation 3.2), which does not represent a real alignment of sequences but denotes the sum of completed match scores, we have two options. The first one is to take the value from the last column so we always keep track of the total score. This leads to the fact that the value in this line will always stay the same or increase, but never decrease. The second choice is taken when a match reaches its maximum and has a minimum score of \( T \). This is the consequence of an ending match meaning this choice is just taken when we calculate the cell for an unmatched region. To summarize the equation for the first row we can say the algorithm choses the maximum value of the last column minus \( T \) or adopts the value of the preceding cell in the first row, depending which one was larger.

Equation [3.3] for the fields aside of the first row is amended by the possibility to begin its alignment score with the value standing in the first row of its own column. Durbin et al. do not mention the fact that this makes it easier for subalignments of long sequences to reach the threshold score once a few matches have been found before. This issue has not been investigated in this work. The total score of the whole alignment is stored in \( F(i + 1, 0) \). This cell lies in an unmatched region of \( x \) and contains the sum of all completed match scores.

\[
F(i, 0) = \max \left\{ \begin{array}{l}
F(i - 1, 0), \\
F(i - 1, j) - T, \quad j = 1, \ldots, m
\end{array} \right.
\]

(3.2)

\[
F(i, j) = \max \left\{ \begin{array}{l}
F(i, 0), \\
F(i - 1, j - 1) + S(x_i, y_j), \\
F(i - 1, j) - g, \\
F(i, j - 1) - g
\end{array} \right.
\]

(3.3)
3.4.2 Match Retrieval

To find similar user behaviour we are only interested in the matches that reached the threshold score. After aligning every user session with each other we extract all matches from those alignments. We consider every subset of the alignment that is not an unmatched region as a match. Therefore, a match consists of two sequences of numbers and can contain gaps. We encode a gap with the value -1. Figure 3.5 shows an example of a match, containing one gap in the first sequence, two exact tasks and one substitution.

Each of those matches is searched in all user sessions. For this, each position in the user
Figure 3.5: Example for a retrieved match. Each of the two subsequences of the sequences the match was generated from is printed in a row.

sessions is checked if the match in question fits at this position. Before we can do this, we have to explain what it means that a match is found since we have to take into account how a match will be transformed into a task later. When a match is found we store the ID of the user session it occurred in as well as its position in the user session. There are some rules that have to be fulfilled to consider a match as found:

**Rule 1**
If both elements of a position in a match are equal, this element has to be found at this position of the subsequence in which we search.

16  4
16  4

*Figure 3.6: Example for rule 1. Will only match a user session subsequence 16 4.*

**Rule 2**
If one element in a match is a gap (-1), the other element of this position is optional at this position.

16  -1  6
16   4  6

*Figure 3.7: Example for rule 2. Will match user session subsequences 16 4 6 and 16 6.*
Rule 3
If both elements are not equal and the next position contains two equal elements (a) or a gap (b), either of the unequal elements can occur at this position.

(a) Example for rule 3 (a). Will match 16 4 and 14 4.

(b) Example for rule 3 (b). Will match 16 4 6, 14 4 6, 14 6 and 16 6.

Figure 3.8: Examples for rule 3 (a) and (b).

Rule 4
If both elements are not equal and the elements of the next positions are not equal as well:

1. Group all elements of one sequence together as long as the next positions still have unequal elements.

2. A match is found if the user session matches completely one of those grouped sequences.

16 18 14
15 17 16

Figure 3.9: Example for rule 4. Match is found if a subsequence of a user session is 15,17,16 or 16,18,14 but not 15,18,14.

Match Sorting Once all matches have been retrieved they have to be sorted to have an order they will be replaced in the sequences. The sorting criteria are the following: First we sort by the number of occurrence in all sequences. The reason for this is that we want to prefer the most occurring match over all other match properties because it is likely that a task with a high occurrence represents the average user behaviour. If two matches have the same occurrence count, the length of the matches is considered. Longer matches will be replaced first. The next two sort properties are introduced for a deterministic behaviour of the algorithm. If two matches have the same occurrence count and the length, the sum of the user session IDs is considered and after that the sum of the task IDs.
3.4.3 Task Generation

After sorting the matches, the actual task is created from each match that has a minimum occurrence of $f$. This parameter is introduced to exclude those matches that just have been performed by a small number of users and therefore are not representative for a general interaction with the software.

**Definition 3.9**

Let $f$ be the number of occurrences a task must have in all user sessions in order to be replaced.

We will now see why we had to search the matches according to the rules enlisted in the previous section. The task generation rules correspond to the search rules. We start with an empty task of temporal relationship type *sequence*. This is the root node of the task we will convert the match to. For this, the procedure is to evaluate each position of the match and apply some rules again. If both elements in the match are equal, the task with this ID is added to the sequence. This is the equivalent to search rule 1. Search rule 2 transferred to task generation leads to the creation of tasks with temporal relationship type *optional*. The created optional task has the task that is "aligned" with the gap as its child. If the algorithm observes a position where search rule 3 is applied, a single selection with two children is created. Both of the two unequal tasks of this position have this selection as their parent. The adopted application of search rule 4 creates three new tasks, a selection and two sequences. The selection is added to the root sequence, both of the sequences are children of the selection. The sequences themselves contain the subsequent unequal tasks of one row of the match. Figure 3.10 shows a match that is converted into a task. In this match all rules have been applied.

3.4.4 Replacement

One last step is required to finish the sequence detection: the replacement of all generated tasks in all the user sessions. For this we create an instance of the generated sequence and replace each subsequence in the user sessions that fits the model of the instance. We do not need to search the positions where we can insert the instance since we stored all the occurrences in our first search. What we have to do is to update the information about the position in all other stored match occurrence data. For example, if we extract six task instances, match occurrences with a position after this replacement need to subtract six from their start and end indices. If a first replacement overlaps another one, the second one is ignored. Here we can see again that the order of replacement is very important because it determines which matches are replaced and which may be ignored.
3.4.5 Repetition

Same as in Harms et al.’s approach the steps iteration detection and sequence detection are repeated until a specific condition is reached. Harms et al. repeat until no further replacements can be made. We propose to stop the repetition before. Possible conditions could be that the algorithms stops once it finds less matches than a specific percentage of the number of matches that have been found in the first iteration of the algorithm. Another possibility would be to have a fixed number of iterations. We will evaluate those proposals in the case study section.

Figure 3.10: Example for task generation. The match on the left is converted into the task on the right.
4 Implementation

S. Herbold and P. Harms implemented their approach in the proof of concept implementation AutoQUEST. AutoQUEST “provides diverse methods for assessing the quality of software. AutoQUEST’s internal algorithms operate on abstract events, which makes AutoQUEST independent of the platform of an assessed software”[6]. Another feature of AutoQUEST is that it is modular and extensible via plugins. AutoQUEST is written in Java 1.7 and is published and licensed under the Apache License Version 2.[1] In this chapter, we will document how we extended AutoQUEST by our task tree generation method.

The alignment sequence detection was implemented in one subversion branch of the autoquest-core-tasktree package. In AutoQUEST this module is responsible for the task tree generation. The most issues we had to solve existed due to the complexity of the problem, which is $O(n^2)$ for most of the utilized algorithms.

4.1 Substitution Matrix

4.1.1 Generation

The generation of the substitution matrix is a step we just need to do once at the beginning of the sequence detection. For a reasonable number of unique tasks no problems were expected but for a large number an efficient storage had to be implemented. Since the substitution matrix is symmetrical it is common practise to just store

$$\frac{n(n + 1)}{2}$$

entries and not the whole matrix. Another improvement is that just a single array of primitive floats has been used which saves space again, namely $n$ times the size of a float array. The generation of the substitution matrix is so far not parallelized but it is possible to do so. The calculation of one cell in the matrix is nearly completely independent from another cell. This would save further time, given enough processors.

4.1.2 Updating

The procedure of updating the substitution matrix is implemented but has just been tested on a very small data set. Since the Matrix grows with $O(n \times (n + 1)/2)$ for large data sets the calculation of non-event-tasks is switched off because each generated task increases n. Also, the computation of non-event-tasks is complex itself. First, for each task all event-tasks have to be searched. This is done by implementing the visitor pattern. Then for each found event-task distances have to be calculated or read from the substitution matrix. To solve the issue of retrieving the event-tasks for each task again they are stored. This process could as well be parallelized but still, storing in the matrix has to be synchronous. Another point is when the matrix is updated, its size has to be enlarged. We could not use dynamic structures like the Java Collection ArrayList because its memory consumption becomes very inefficient when storing a large number of float values. The solution was to keep using simple arrays and reserve more memory than initially needed. So if the substitution matrix update is switched on, the matrix preallocates 10 times the initial unique tasks and doubles each time the size limit is reached. This increase of the size is very expensive since the whole matrix has to be copied to the newly created, larger sized matrix.

4.2 Alignments

Currently available and popular alignment tools are usually developed for the use in a biological context. The problem with these is that those tools are just working on sequences with a limited alphabet like the bases in DNA (A,C,T,G), RNA (A,C,U,G) or polypeptides (alphabet size 20). Because of this issue we could not reuse well tested implementations of common or more complex algorithms.

The Smith Waterman algorithm and its modified version are $O(n^2)$ algorithms with $n$ being the length the sequences being aligned. This may not be an issue because the user session itself may not be too long. The actual problem is that we have a large number of sequences which we have to align against each other. We solve this issue by dividing the workload to as much pieces as processors are detected. Each piece consists of a series of user sessions e.g. user session 1-10, 11-20,… and so on. For each piece we create a thread that computes the assigned alignments with all other user sessions and stores the extracted matches into a shared data structure.

4.3 Match Search, Sorting and Replacement

Like the alignment process, the search, sort and replace steps take a lot of time and have to be performed in each iteration of the algorithm. Again, the solution to those problems is
parallelism. The collection of all matches is divided into smaller collections and assigned to a worker thread. These threads search "their" matches in all user sessions.

The sorting routine is the standard Java Collections sort method. This function does not perform a parallel search, which could speed up the algorithm again. In Java 8 a new Sort API has been introduced, which allows parallel sorting on arrays. The implementation would benefit from using this API but so far AutoQUEST is written for Java 7.

The replacement step is divided in two steps. First all replacements are planned, meaning they are added to a user session specific queues. Those queues contain all replacements that will be performed in this user session in their correct order. Once the replacements are planned all further steps can again be parallelized well, gaining a huge speedup of the whole algorithm.
5 Case Study

To validate our method, we run the alignment approach of task tree generation against the same case study as Harms et al. did. The user traces were collected on an application portal of the university of Göttingen. Figure 5.1 shows a screenshot of the first page of the portal. After logging in, users can fill out multiple forms regarding their personal data as well as upload their CVs. In this case study 555 user created around 3602 appropriate user sessions. Further details about the data are enlisted in table 5.1. The data of the traces is XML structured. Figure 5.2 shows the format of one onclick event.

Master Application Portal  Applied Computer Science

Welcome to the Applied Computer Science online application form!

The Applied Computer Science is a M.Sc. program offered by the Georg-August-University Göttingen. To apply for the Master's program you need to fill in the online application form. You may want to gather some information before you begin your application, but you can make a start without this and save your form part way through. Please check the program specific requirements and a list of required documents on the following page: Application for the Applied Computer Science MSc.

To begin your application you will need to create a user account. To do this please click on “Create new account” button. If you have already started your online application just enter your username/password pair and click on the “Login” button.

Username*  
Password*  

Click here if you forgot your password.

Login  Create new account

Figure 5.1: Screenshot of the website the data was collected with.
Figure 5.2: Example of one event in XML structure.

<table>
<thead>
<tr>
<th>Date, Users &amp; Sessions</th>
<th>Start of Recording</th>
<th>25 October 2013</th>
</tr>
</thead>
<tbody>
<tr>
<td>End of Recording</td>
<td>7 March 2014</td>
<td></td>
</tr>
<tr>
<td>Recorded Users</td>
<td>555</td>
<td></td>
</tr>
<tr>
<td>Recorded User Sessions</td>
<td>4,129</td>
<td></td>
</tr>
<tr>
<td>Considered User Sessions</td>
<td>3,602</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Events</th>
<th>Recorded Events</th>
<th>350,368</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevant Events</td>
<td>306,568</td>
<td></td>
</tr>
<tr>
<td>Double Clicks</td>
<td>6,437</td>
<td></td>
</tr>
<tr>
<td>Focus Changes</td>
<td>89,825</td>
<td></td>
</tr>
<tr>
<td>Considered Events</td>
<td>210,306</td>
<td></td>
</tr>
<tr>
<td>Different Events</td>
<td>1,897</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Case study overview.
We are interested in several aspects of the task tree creation with the alignment approach. First of all, we want to examine the necessity of the calculation of the distance between non-event-tasks. The calculations of those distances are very expensive operations so we want to know if this step could be left out and if we still find approximately the same amount of tasks in the same quality. After that, we study the algorithms termination conditions with the goal is to find out if we can terminate the algorithm earlier than proposed in chapter 3.

Afterwards, we evaluate the performance of the alignment approach and compare it to the performance of the n-gram approach. At last, we discuss the created task tree by means of representative task examples after running the alignment approach on the full case study. All experiments were executed on an AMD Opteron(TM) Processor 6276 (64 core) with 250GB of ram after we figured out that an Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz with 8GB of ram is not sufficient to run the alignment approach on large numbers of user sessions. The Java virtual machine was started with the following parameters:

-XX:+UseConcMarkSweepGC
-Xmx84000m

### 5.1 Parameters

The alignment approach has several parameters that have to be set before running the algorithm. Some values of the parameters may also depend on the underlying data. All parameters are described in the approach chapter. Table 5.2 gives a short summary of all available parameters that can be set and also the values we used in this case study. Those parameters were manually chosen by trial and error and do not guarantee the best possible results.

### 5.2 Data Preprocessing

After loading the input data from all XML files the following preprocessing steps are performed. The information about each command is copied from its manual page in Auto-QUEST. The goal of the preprocessing is to fix several flaws the input data has.

**condenseHTMLGUIModel**
Merges all equal nodes in the GUI-Model.

**condenseMouseClicks**
Reduces a sequence of mouse button down, mouse button up and mouse click with the same button on the same event target to a single mouse click with that button on that target. The mouse button down and mouse button up events are discarded.
5. Case Study

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Function</th>
<th>Definition</th>
<th>Value in case study</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>The value of the maximum score in the substitution matrix</td>
<td>3.5</td>
<td>6</td>
</tr>
<tr>
<td>$L$</td>
<td>Penalty for the score between non-event-tasks</td>
<td>3.6</td>
<td>3</td>
</tr>
<tr>
<td>$T$</td>
<td>Threshold score for determination of match importance</td>
<td>3.7</td>
<td>9</td>
</tr>
<tr>
<td>$g$</td>
<td>Gap penalty for inserting gaps</td>
<td>2.1</td>
<td>3</td>
</tr>
<tr>
<td>$f$</td>
<td>Number of occurrences a task must at least have in all user sessions to be replaced</td>
<td>3.9</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.2: Table of all parameters of the alignment approach that can be adjusted for the task tree generation.

**correctKeyInteractionTargets**
Iterates the provided sequences and sets the target of all key interaction events to the GUI element having the current keyboard focus. The current keyboard focus is determined either by keyboard focus events or by using the target of the first key interaction in a sequence. Events changing the keyboard focus are discarded herewith.

**correctTabKeyNavigationOrder**
Iterates the provided sequences and corrects the order of events in case of tab key navigation. This is required, as from time to time the event of pressing the tab key for navigation in formulars comes before the text input event in a text input field out of which the tab key navigates.

5.3 Calculation Of Distances Between Non-Event-Tasks

In this section we investigate if the calculation of the distances between non-event-tasks is necessary by comparing the two generated task trees, one with the additional calculations and one where we set every score involving a non-event-task to zero. Table 5.3 shows the results of this experiment. The calculation of distances between non-event-tasks is not an issue on very small subsets (ca. 40 user sessions) of the case study. But once we include more sequences the index for the array we use for storing the scores in the substitution matrix hit the limit of `Integer.MAX_VALUE`. A possible solution for this issue is not to calculate the distances between the non-event-tasks and thus saving computation time and memory usage. The number of found tasks is significantly higher when the distances between non-event-tasks are calculated but the drawback is that the time to generate the task
The quality of the generated task trees also differs. One criterion of the quality of task trees is the depth. The depth is defined as that each level in a task tree increases the depth by one. Without the calculated distances the generated task trees have a flatter structure (maximum depth is 6 with calculated distances, 4 without). This flat structure can be explained by the reduced number of repetitions of the algorithm because we hit the termination condition earlier in the algorithm. An early stop decreases the possibility of inserting new levels into the tasks trees.

With the calculation of the distances enabled longer interactions like login or account creation procedures can be found. Those long tasks do not always represent correct user interactions. Figure 5.3 shows a task tree for the first part of the account creation process.

Figure 5.3: An example for a task found by alignment task tree generation with distance calculation between non-event-tasks.
We can see several subtasks of the task where the subtask does not model user behaviour correctly. Selection 2691 followed by optionality 2692 makes no sense since the text input of the optionality should just happen after the text input field has been clicked on. This happens in iteration 1288, the first child of selection 2691. The optionality with the text input should actually be in a sequence with iteration 1288 as its preceding element. Another part of the task tree that represents no real user behaviour in this example task is sequence 1656. The first selection in it gives the possibility to either click or double click on an input field. While double clicking a text field is not an effective behaviour of a user, it is still a valid action to achieve his goal to enter his email address. The meaningless part is the text input on the email input field, followed by the same event again. This should either be just one event at all or be found in an iteration. There are several more examples where the non-event-task distance calculation did not improve the task tree quality. In summary, the amount of tasks created but not their quality can be increased by enabling the distance computing.

With the large increase of computational time and the integer limitation of the Java array indexes in mind, we will set the score to or from non-event-tasks to zero in all further experiments. The reason for this is that we have an increase in time by over 3800% even in this very small example and we cannot assume a linear scaling of this increase because nearly all algorithms we use have a complexity of $O(n^2)$. A parallel computation of the distances and a clever storage of all already calculated values could as well fix this issue but this could not be addressed in this thesis due to time issues.

<table>
<thead>
<tr>
<th>Testcase</th>
<th>Type of non-event-task</th>
<th>with distances</th>
<th>without distances</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequences</td>
<td>352</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Iterations</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>Selections</td>
<td>328</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Optionals</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

**Performance indicator**

<table>
<thead>
<tr>
<th></th>
<th>Time (s)</th>
<th>Number of repetitions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>72.2</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>1.9</td>
<td>7</td>
</tr>
</tbody>
</table>

*Table 5.3: Results of the version with and without calculation of the distances between non-event-tasks*
5.4 Evaluation Of Termination Conditions

In this section we will investigate if the algorithm for the task tree generation can be terminated with other conditions than the condition mentioned in chapter 3. The described behaviour is that the algorithm stops if no further replacements could be performed. It has been observed that the most matches are found in the first few repetitions of the sequence detection phase. Table 5.4 and the corresponding figure 5.4 supports this claim. We can see that the number of matches found in each repetition decreases rapidly. After the 6th repetition we detect 0.01% of the matches found in the first sequence detection. We could stop the algorithm here. But since the algorithm finishes shortly after (after the 10th repetition), we keep the termination condition for this case study as it is defined in the approach. If on other case studies the algorithm repeats more often, it can be considered to change the termination condition to a fixed number of iteration. Another probable condition is to stop when the number of found matches falls below a fraction from the matches found in the first repetition.

As we see in section 5.3 the number of iterations drastically increases if the distances between non-event-tasks are calculated. If a good solution for the large computation time of the distances is found, this would be a possible use case of a different termination condition.

<table>
<thead>
<tr>
<th>Repetition No.</th>
<th>Matches</th>
<th>Absolute</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1,112,794</td>
<td>26.17</td>
<td>26.17</td>
</tr>
<tr>
<td>1</td>
<td>381,190</td>
<td>39.16</td>
<td>12.99</td>
</tr>
<tr>
<td>2</td>
<td>167,677</td>
<td>45.52</td>
<td>6.36</td>
</tr>
<tr>
<td>3</td>
<td>63,964</td>
<td>51.00</td>
<td>5.48</td>
</tr>
<tr>
<td>4</td>
<td>26,171</td>
<td>55.83</td>
<td>4.83</td>
</tr>
<tr>
<td>5</td>
<td>12,627</td>
<td>60.72</td>
<td>4.89</td>
</tr>
<tr>
<td>6</td>
<td>8,660</td>
<td>65.48</td>
<td>4.76</td>
</tr>
<tr>
<td>7</td>
<td>7,517</td>
<td>70.34</td>
<td>4.86</td>
</tr>
<tr>
<td>8</td>
<td>7,232</td>
<td>75.02</td>
<td>4.68</td>
</tr>
<tr>
<td>9</td>
<td>7,214</td>
<td>79.74</td>
<td>4.72</td>
</tr>
<tr>
<td>10</td>
<td>7,214</td>
<td>84.49</td>
<td>4.75</td>
</tr>
</tbody>
</table>

Table 5.4: Matches found per repetition and the time for each repetition. All times in minutes.
### 5.5 Performance Evaluation

To evaluate the performance of both the alignment approach and the n-gram approach we run each task tree generation algorithms against different sized subsets of the case study. We chose user session sets with the size of 10, 100, 1000 and the full case study (3602). Figure 5.5 and table 5.5 show the results of this experiment. We can see that the alignment approach is slower on any size of input data. But with the increasing number of user sessions we can observe that the computation time increases notably. As a side note the memory consumption of the alignment approach for the full case study was 46.9 GB at its maximum. The n-gram approach used 1.3 GB memory for the same data set. The memory usage of the other data set sizes were not tracked. In summary, the alignment approach is definitely more resource intensive than the n-gram approach.

![Graph with matches found per repetition and the time per repetition.](image)

*Figure 5.4: Graph with matches found per repetition and the time per repetition.*
Table 5.5: Comparison between execution times (seconds) of n-gram approach and alignment supported task tree generation. Note that the execution time of the alignment approach for 10 user sessions seems to be an approximation of \( \pi \).

<table>
<thead>
<tr>
<th>Number of user sessions</th>
<th>n-gram approach</th>
<th>alignment approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2.655</td>
<td>3.141</td>
</tr>
<tr>
<td>100</td>
<td>5.681</td>
<td>12.019</td>
</tr>
<tr>
<td>1000</td>
<td>60.77</td>
<td>419.063</td>
</tr>
<tr>
<td>3602</td>
<td>1,417.370</td>
<td>5,780.050</td>
</tr>
</tbody>
</table>

Figure 5.5: Scaling behaviour of both approaches.
5 Case Study

Table 5.6: Comparison of the n-gram approach from Harms et al. and the alignment approach. The alignment approach finds less task but is able to detect selections and optionals.

<table>
<thead>
<tr>
<th>Type of task</th>
<th>Number of tasks n-gram approach</th>
<th>alignment approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>All non-event-tasks</td>
<td>10,634</td>
<td>4,635</td>
</tr>
<tr>
<td>Sequences</td>
<td>9,530</td>
<td>2,759</td>
</tr>
<tr>
<td>Iterations</td>
<td>1,104</td>
<td>619</td>
</tr>
<tr>
<td>Selections</td>
<td>-</td>
<td>1,156</td>
</tr>
<tr>
<td>Optionals</td>
<td>-</td>
<td>101</td>
</tr>
</tbody>
</table>

5.6 Generated Task Trees

In this section we analyse the tasks we generated with the alignment approach. We were able to create tasks that describe effective user behaviour but also could find examples for wrong tasks. With wrong tasks we mean tasks that allow actions that are either not possible on the underlying GUI-model or make no sense. The quantity of created tasks can be found in table 5.6. We find around 6000 tasks less than Harms et al. but are able to create tasks of temporal relationship type selection and optional.

The created tasks in the alignment approach differ from those of the n-gram approach in quality as well. Figure 5.6 shows a task tree that consists of two different interactions: A login and a password reset. The task starts with the tab key pressed on a textfield that is for entering user names. We can see that the actual entering of the username follows at the end of this task tree. First, the textfield for the user name is clicked on in iteration 210306 and then input is entered in iteration 210312. The task tree should have iteration 210306 as the first element in sequence 2098703. The wrong order of some interactions can be observed in other examples, too.

Another notable point in this task tree is that it combines two different user interactions. The input of the email address is a text input field of the page where users can reset their password while the login button allows users to enter the personal area after entering their credentials. It is not correct to merge both tasks into one task tree since they are on different pages. The alignment approach identifies the different actions correctly as a selection (selection 2098704) though.

The alignment approach for task tree generation creates tasks like shown in figure 5.7. It is basically a selection of two sequences, each sequence has two event-tasks. The order of the event-tasks in one sequence is reversed in the other sequence. We find numerous tasks of this kind.

The main issue with tasks like this is that a ValueSelection on an inputCheckbox and
a click on this checkbox are just one action by the user but two events are created for this action. This task is an example for the necessity of further preprocessing of the input data. Similar as in the condenseMouseClicks preprocessing command, all clicks on an input_checkbox before or after a value selection should be discarded.

The last issue we address in this section is the missing detection and merging of similar or equal tasks. Figure 5.8a and figure 5.8b both describe the same user interaction of setting a new password. The tasks themselves represent correct user behaviour and are positive examples for tasks generated by the alignment approach. The only difference is the order of the children in selection 2611158. But since the order in a selection does not matter, both generated task trees are semantically equal.
5 Case Study

**Figure 5.7:** Example for a created task that shows that further data preprocessing is required.

```plaintext
sequence #210925
  selection #210926
  sequence #210927
    ValueSelection("on") => input_checkbox(id="id_is_fulltime") #118
    LeftMouseClick => input_checkbox(id="id_is_fulltime") #117
  sequence #210928
    LeftMouseClick => input_checkbox(id="id_is_fulltime") #117
    ValueSelection("on") => input_checkbox(id="id_is_fulltime") #118
```

**Figure 5.8:** Example for a task for setting a new password.

```plaintext
sequence #2611157
  selection #2611158
    KeyPressed TAB => input_password(id="id_new_password1") #645
  iteration #210774 (LeftMouseClick => input_password(id="id_new_password2") #661)
    LeftMouseClick => input_password(id="id_new_password2") #661
    TextInput => input_password(id="id_new_password2") #647
(a) Version 1

sequence #1180421
  selection #1180422
  iteration #210774 (LeftMouseClick => input_password(id="id_new_password2") #661)
    LeftMouseClick => input_password(id="id_new_password2") #661
    KeyPressed TAB => input_password(id="id_new_password1") #646
    TextInput => input_password(id="id_new_password2") #647
(b) Version 2
```

*Figure 5.8: Example for a task for setting a new password.*
5.7 Discussion

The results of all experiments show that the alignment approach still has to cope with some problems. First of all, the performance and resource usage are a downside of this approach. The alignment version of autoquest is no longer able to run its task tree generation on common bulk hardware. Even on a server with 64 processors and a large amount of memory our approach is still slower than the n-gram approach implementation, which does not use any parallelisation. When calculating distances between non-event-tasks the expected performance is even worse, although it may be worthy calculating those.

Another point is that the alignment approach is not able to identify and merge similar tasks. This problem has also not been solved by Harms at al. yet. There are several ways or metrics on how to determine if two tasks are equal. For example, we could calculate the Levenshtein distance of two tasks. The calculation of Levenshtein distance is very similar to the Needleman-Wunsch and Smith-Waterman algorithm. It uses a very simple scoring scheme as it just counts the number of insertions, deletions or mismatches between two sequences. Once two equal tasks are found they should be merged. We do not propose an algorithm for merging tasks in this work.

In our case study we also detected that more preprocessing of the input is needed in some cases in order to generate better task trees. Clicks on an input-checkbox before or after a ValueSelection should always be discarded. Further research is needed to find other similar necessary preprocessing steps. An imaginable preprocessing function could be to remove clicks on radio button, too if ValueSelections on that radio button pre- or succeed the click.

The last problem we discuss in this section is the wrong order of tasks in a task tree and its consequence of incomplete task trees. The task trees we generate are highly dependent on the scores we assign to different components such as substitution matrix and the gap penalty. The scores we calculate for substitutions in our substitution matrix are just one possibility of a metric of task similarity. The distance in a GUI-Model is a good start to find similar or related tasks but may not be sufficient to completely model user interactions on specific graphical user interfaces. Further research is needed to improve the scoring for substitutions. For example, it could be possible to manually create a database of tasks that are usually related to each other. This database could be used to add more knowledge to the substitution matrix.

It is also interesting to find improvements for the values of the gap penalty, the maximal score of the substitution matrix and the alignment threshold score. Further research could focus on finding an optimal combination of all parameters to improve the amount and the quality of the generated task trees.
6 Conclusion

With the alignment approach for sequence detection we created a valuable framework for task tree generation. It is able to detect common interactions with a software of multiple users, to weight the importance of each found interaction and to create a task tree that represents the users behaviour. For the creation of the task tree we use methods adopted from bioinformatics that are able to perform an approximate comparison of sequences of tasks. This alignment method is well configurable by several parameters that influence the quality and amount of created task trees. Disadvantages of this approach are performance and resource intensity issues as well as a missing merging of similar task trees.

Further research should focus on the evaluation of the alignment and substitution matrix parameters. It is conceivable that those parameters can be automatically learned or adjusted to the processed data. This may allow to find good results on any kind of data. For the learning of the parameters we would need to have a definition of what a ideal task tree for a specific task looks like. This leads to the overall question what a good task is in the perspective of usability. If such a definition existed, one could optimize the parameters to create a task tree that converges to the ideal task tree.

In summary, we can say that once the automatic adjustment of the alignment and substitution matrix parameters is researched the overall goal to have a method for semi-automatic usability evaluation can be achieved.
Bibliography


Acknowledgements

I would like to offer my special thanks to Patrick Harms who guided me through this thesis. I also wish to acknowledge the proof reading done by Hanna Holderied, Christian Müller, Sarah Schüßler and Jonas Krimmel. Additionally my thanks are extended to my colleagues Piotr Kasprzak and Daniel Adler of GWDG for supplying me with powerful hardware and programming advices.