Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

iOS Swipe

Bachelor Thesis

Timo Bräm

braemt@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Philipp Brandes, Laura Peer
Prof. Dr. Roger Wattenhofer

August 21, 2016

Acknowledgements

I thank the members of the Distributed Computing Group, my family and friends for the participation in the evaluation and for the feedback.

Abstract

There exist many onscreen keyboards for smartphones, but only a few of them are capable to handle a swipe, i.e. the possibility to write a word without raising the finger. In this thesis we classify the swipes based on a dictionary. Many concepts are designed to achieve real time performance. The implementation of the keyboard focuses on Apple's operating system iOS. The evaluation shows that the probability that the swiped word is suggested, i.e. the swiped word appears in one of four words in total, is 96%.

Contents

Chapter 1

Introduction

1.1 Motivation

Several years ago, when the first cellphones appeared, typing an SMS was hard work. You had to press a key up to four times to write one single letter. With the emergence of the smartphones typing a word got a lot easier. Now every character has its own key. This is not the end of the journey, nowadays you can write a word without raising your finger. It is called swiping and is shown in Figure 1.1. A word is swiped by moving the finger over the screen from one letter to the next in one stroke. This is very convenient to use, because the user does not have to hit every letter perfectly and hence he can write faster. Words with double letters are swiped as if they had no double letters, i.e. the words 'of' and 'off' have exactly the same swipe. The same concept is applied for special characters. For example, 'it's' has the same swipe as 'its'. The goal of this thesis is to find a way to develop a keyboard for iOS that supports swiping, even if the user does not always perfectly hit the letters he intended to swipe. This will be done by looking through a dictionary and find the word that matches best. Furthermore, the swipe feature will be integrated in an existing keyboard that adapts to the user's writing style.

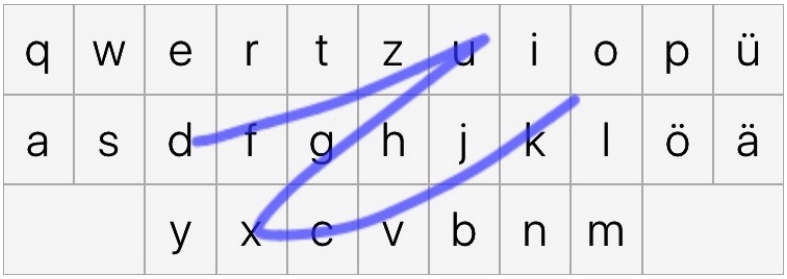

Figure 1.1: The path that is drawn by a user swiping the word 'duck'. The user did not raise his finger until he reached the letter 'k'.

1.2 Related Work

Swiping is a text input method whose implementation has several different approaches. The implementation of our keyboard is centred around the Dynamic Time Warping algorithm, which is a well known algorithm for measuring the difference between two temporal sequences. This approach was also mentioned in the paper from Shumin Zhai and Oer Ola Kristensson [1], where a keyboard was presented that supports swiping with a stylus instead of the user's finger. Smith, Bi and Zhai [2] investigated how it is possible to lower the error rate caused by the ambiguity of swipes on the QWERTY keyboard. For this they looked at different keyboard layouts.

There are some popular keyboards on the market that support swiping, for example Swype [3], SwiftKey [4] or Gboard [5]. These keyboards were developed by big companies and improved over time and therefore they are on a high technical level. Besides typing and swiping, the Gboard keyboard offers various features, for example searching and sending videos, weather forecasts or sport scores.

The Kännsch keyboard for iOS was presented in the bachelor thesis of Andres Konrad [6]. It is a predictive keyboard that adapts to the user's writing style. Besides the German and English dictionary, Kännsch also supports a Swiss German dictionary that evolves as the user writes. As part of our thesis we expanded Kännsch such that it supports typing and swiping simultaneously.

Chapter 2

Design

A user who writes a word on the keyboard without raising his finger produces a swipe. The swipe is a list of points (coordinates on the keyboard) tracking the user's finger. Later in this chapter we will introduce an algorithm that measures how likely it is for a swipe to belong to a given word. Because this algorithm is costly, the next section explains a method how to reduce the work for algorithms of this kind.

2.1 Filters

Assume that there is an algorithm that takes a swipe and a list of words and returns the word that matches best to the swipe by looking at every word in the list. This algorithm can be made faster by shortening the list of words. There are four such shortening techniques, referred as filters, that are used in our keyboard. The order the filters are applied is the same as they are presented in this section. That means, for example, that the second filter has to process much less words than the first filter, because only the words that satisfy the criterions of the first filter are passed to the second filter. How these filters affect the runtime and the quality of the output will be evaluated in Chapter 4.

2.1.1 Start Point

If a user starts to swipe on the right side of the keyboard, then he most probably did not intend to swipe a word that starts with the letter 'a'. In fact, if a user wants to swipe a word, he tries to start at the location of the first letter of the word. There is some variance in the precision of the user's finger that has to be taken into account and therefore more than one letter should be considered as the first letter of the swiped word as shown in Figure 2.1. This filter is stable in the sense that for every swipe at most four letters are considered as potential first letters of the swiped word. This gives an upper bound for the number of filtered words, i.e. the words that are included in the wordlist. In the English

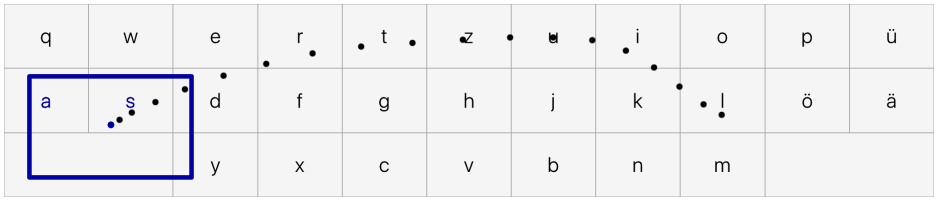

Figure 2.1: Example of a swipe of the word ‘still’. The first point in the swipe defines the area with the potential first letters of the swiped word. In this case these are only the letters ‘a’ and ‘s’, because the centre of these two keys are in the blue area. That means ‘still’ is included in the list of potential swiped words.

dictionary [7] there are at most 36′857 out of 160′606 words included (22.95%). This is the case when the swipe starts between the letters ‘w’, ‘e’, ‘s’ and ‘d’. With the start point filter the dictionary is reduced as soon as the user started the swipe. The runtime of this filter is constant, because for each letter in the alphabet one check is sufficient to determine whether this character lies in the start point area. All words with the same first letter are stored together in an array and therefore we only need to reference the corresponding arrays.

2.1.2 Start Direction

Another way to filter the dictionary right from the beginning of the swipe is to consider the line that goes through the start point and a following point of the swipe and then include all words in the wordlist whose expected start direction does not deviate too much from the actual direction given by that line. In Figure 2.2 the line goes through the first and the fifth point of the swipe. The following point, in this case the fifth point, is chosen such that the length of the swipe between the two points is about the same as the distance between two neighbouring letters on the keyboard. This is because the following point should not be too near to the start point to improve accuracy and it should not be too far away, because the swipe should not make a turn until it reached that distance. However, if a word starts with two letters that are neighbours on the keyboard, for example ‘salt’ as in Figure 2.3, the user tends to make only a very small stroke in the expected direction and then turns around. The direction that is measured does not match the expected direction of the word the user intended to swipe. Therefore, all words that start with two letters that are neighbours on the keyboard get included besides the ones with an acceptable expected start direction. This filter has a big variance in the number of words that are included. In Figure 2.2 almost all words are included while in Figure 2.3 only few words are filtered.

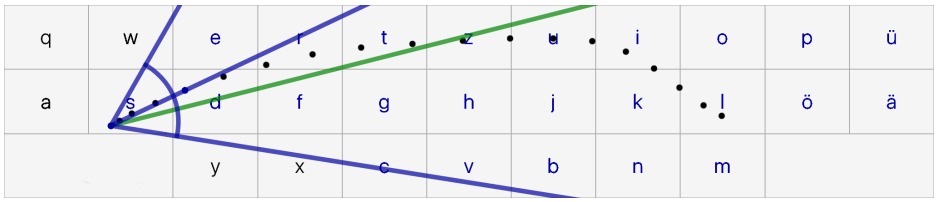

Figure 2.2: The line from the first through the fifth point of the swipe indicates the actual start direction. All words whose expected start direction is in the blue area are filtered. The expected start direction of the word ‘still’ is indicated as the green line. Since this line is in the blue area, ‘still’ is included in the wordlist. Note that the green line is parallel to the line that goes through the letters ‘s’ and ‘t’ on the keyboard.

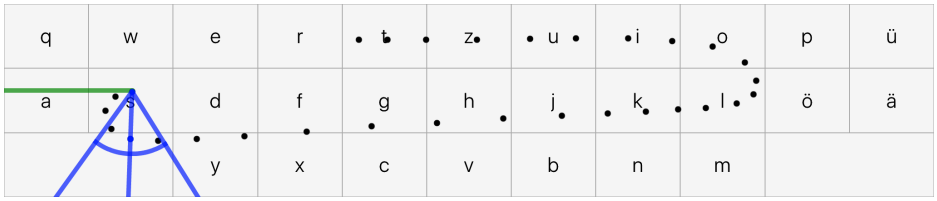

Figure 2.3: Example of a swipe of the word ‘salt’. The blue area does not include the expected start direction of the word ‘salt’ (indicated as the green line).

2.1.3 End Point

The end point filter only takes the last point of a swipe into account. This filter is closely related to the start point filter, i.e. only the words are filtered whose last letter is near to the last point of the swipe (Figure 2.4). It has two main differences to the start point filter. First, the user tends to be more inaccurate and hence there are more letters to consider and second, this filter can only be applied after the user finished his swipe and therefore the words have to be filtered while the user is waiting for the results of his swipe.

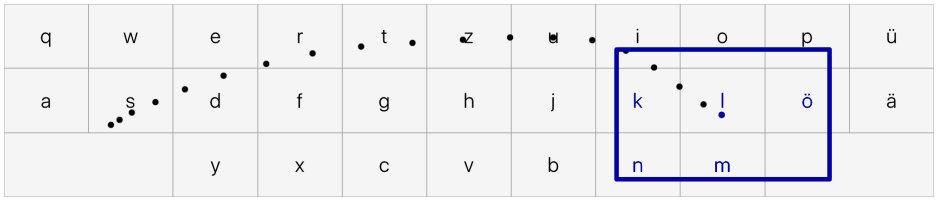

Figure 2.4: Example of a swipe of the word ‘still’. In this case all words are included in the wordlist whose last letter is either ‘k’, ‘l’, ‘ö’, ‘n’ or ‘m’.

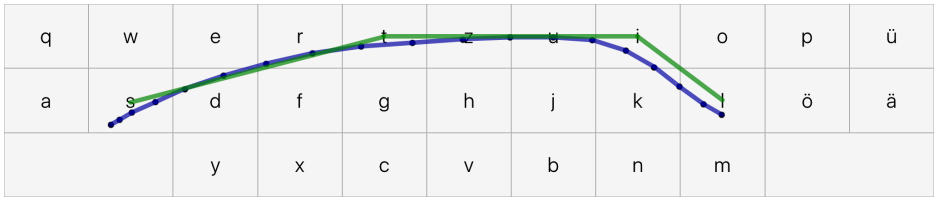

Figure 2.5: The length of the green path is the expected length of the swipe of the word ‘still’. The length of the blue path is the actual swiped length. This length determines which words are filtered by the length filter.

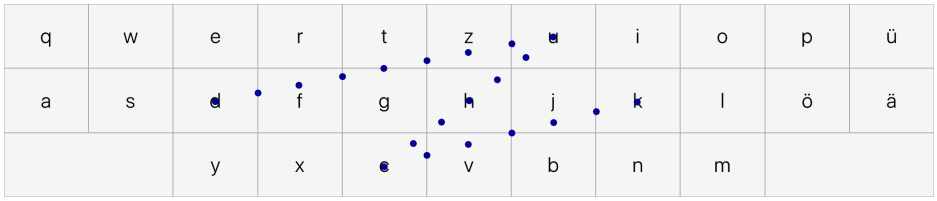

Figure 2.6: The blue points are the expected swipe of the word ‘duck’.

2.1.4 Length

The length of a swipe is the sum of the euclidian distance between all following points of the swipe, in Figure 2.5 this would be the length of the blue path. This length ℓ is compared to the expected length of a given word $\ell(w)$, that is the length of the green path in Figure 2.5 for the word ‘still’. That means every word that satisfies $A \cdot \ell \leq \ell(w) \leq B \cdot \ell$ gets included in the wordlist, where A and B are real constants. The analysis has shown that $A = 0.75$ and $B = 1.5$ are values that work well. Swipes of short words like ‘in’ or ‘was’ often times do not fulfil this criterion. Therefore, an additional constant $C \geq 0$ was introduced that also filters all words that satisfy $\ell - C \leq \ell(w) \leq \ell + C$.

2.2 Dynamic Time Warping

Dynamic time warping (DTW) is an algorithm to measure the difference between two temporal sequences, for example two swipes. By mapping a word to the expected swipe, i.e. the swipe that is going from one character straight to the next as illustrated in Figure 2.6, DTW is a measurement of how likely it is for a swipe to belong to a given word.

user swipe \ expected swipe	$(3,4)$	$(3,5)$	$(2,6)$	$(1,4)$	$(2,2)$
$(4,4)$	+1	+2	+8	+9	+8
$(2,7)$	+10	+5	+1	+10	+25
$(1,4)$	+4	+5	+5	+0	+5
$(0,3)$	+10	+13	+13	+2	+5

Table 2.1: This is the DTW table with the costs between every pair of points of the two swipes. A path from the top left corner to the bottom right corner is searched that minimises the added costs.

2.2.1 Basic Algorithm

The DTW in his basic form takes two swipes (lists of points) and outputs the minimal cost of the mapping of one swipe to the other. This is done by setting up a table with the costs between every pair of points (Table 2.1). Then, a path is searched that goes from the top left to the bottom right and minimises the added costs. Since we are not interested in the path itself, but only in the costs, this can be done by filling out the table t row by row. For each entry in t the path with the minimal added costs up to this point is calculated. This can be solved with the subproblems that were already calculated and we get

$$t[i,j] = c(s_0[i], s_1[j]) + \min(t[i-1,j], t[i,j-1], t[i-1,j-1]),$$

where s_0 and s_1 are the two swipes and c denotes the costs between the two points. In our keyboard the cost function $c : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ between two points is chosen as the squared euclidian distance, i.e. $c(x_1, x_2) = \|x_1 - x_2\|_2^2$. The resulting table looks like Table 2.2. Algorithm 1 shows the pseudo code of the DTW. The runtime is given by $O(NM)$, where N denotes the number of points in the first swipe and M the number of points in the second swipe.

2.2.2 Improved Algorithm

The more points the expected swipe contains, the better the basic algorithm works. That is because a point of the user's swipe may lay in the middle of two points of the expected swipe (Figure 2.7) and therefore the cost is higher than it should be. The downside of having more points in the expected swipe is the higher runtime of the algorithm. This is the reason an algorithm was developed that minimises the entries in the expected swipe and still works almost

user swipe \ expected swipe	(3, 4)	(3, 5)	(2, 6)	(1, 4)	(2, 2)
(4, 4)	1	3	11	20	28
(2, 7)	11	6	4	14	39
(1, 4)	15	11	9	4	9
(0, 3)	25	24	22	6	(9)

Table 2.2: This is the calculated DTW table. Each entry denotes the best path up to this point. The last entry is the final cost of the mapping.

Algorithm 1 Basic Dynamic Time Warping Algorithm

```
function BasicDTW(s_0, s_1)
    M ← |s_0|
    N ← |s_1|
    dtw ← Array[0...M][0...N]

    for i = 1...M do
        dtw[i][0] ← ∞
    end for
    for j = 1...N do
        dtw[0][j] ← ∞
    end for
    dtw[0][0] ← 0

    for i = 1...M do
        for j = 1...N do
            cost ← c(s_0[i − 1], s_1[j − 1])
            dtw[i][j] ← cost+min(dtw[i − 1][j], dtw[i][j − 1], dtw[i − 1][j − 1])
        end for
    end for

    return dtw[M][N]
end function
```

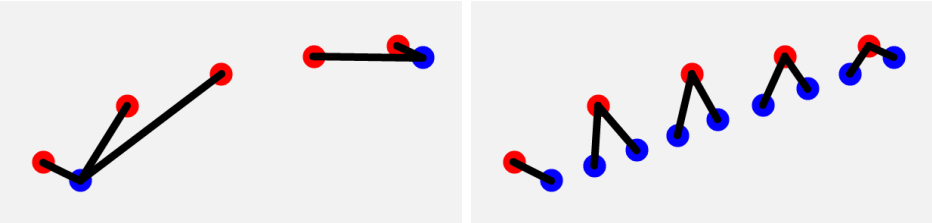

Figure 2.7: The mappings are more accurate as the number of points in the expected swipe (blue points) increases.

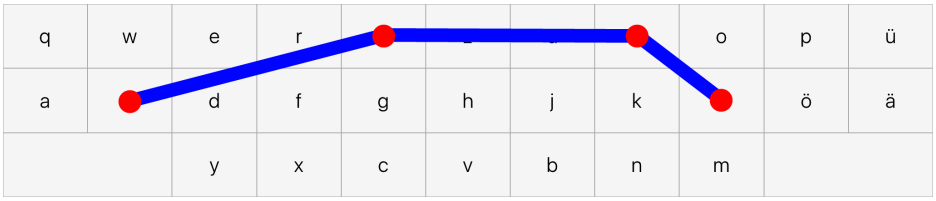

Figure 2.8: Example of the perfect swipe (red/blue segments) of the word ‘still’.

as good as with the expected swipe that contains a lot of points. To achieve this, the expected swipe is no longer represented as a list of points, but as a list of segments and it is called perfect swipe (the user’s swipe is still represented as a list of points). The perfect swipe contains the segment from the first letter to itself, the segment from the first to the second letter, the segment from the second letter to itself and so on until it finally contains the segment from the last letter to itself as illustrated in Figure 2.8. The cost function c of Algorithm 1 is adapted such that it works with segments. An example for the new cost function is the smallest distance between the point from the user’s swipe and the segment from the improved swipe squared.

There is still one issue with this algorithm. In the word ‘still’ there is a segment from the letter ‘t’ to ‘i’. A user who swipes back and forth between those two letters (writing ‘stitititill’) produces a swipe that has the same costs as a swipe that only goes from ‘t’ to ‘i’ once. This problem is solved by saving the location on the segment where the current point of the user’s swipe is mapped to. Every following point has to be mapped between the saved location and the end point of the segment. This is a fast approximation of the basic DTW with a lot of points in its expected swipe.

2.2.3 Locality Constraint

As seen in Section 2.1.4 with the length filter, only the words whose expected swipe length does not deviate too much from the actual swipe length get filtered. The same concept is applicable in the DTW algorithm. Instead of calculating

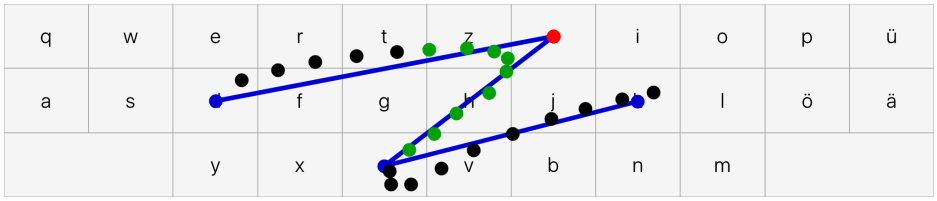

Figure 2.9: Consider the red segment on the letter 'u' of the perfect swipe. The black points of the user's swipe do not satisfy the locality constraint and therefore the costs are not calculated. The green points are in the range where the mappings are possible.

the costs between every pair of segments and points, only the costs between the segments and points are calculated whose length do no deviate too much so far. Here the parameters are chosen in a more conservative way, such that the quality of the predictions is only influenced as little as possible. Figure 2.9 shows that for a given segment of a swipe only the costs of a few points are calculated.

2.3 Input

Depending on the speed the user swipes, there are more or less points in the user's swipe for the same path. That is because the operating system is limited in the speed of reporting the points. The runtime of the improved algorithm depends on the number of points in the swipe. Therefore, a slow swipe with a lot of points is shortened such that it is similar to a fast swipe with less points. This is implemented by looking at three points: the last point that was last added to the swipe (A), the last point that was reported (C) and a point that is in between the other two points (B). There are three criterions and if at least one of them is satisfied, then point B gets added to the swipe (Figure 2.10).

1. No point was added to the swipe for a long time or the swipe contains less than 2 points.
2. The area of the triangle ABC is bigger than some threshold.
3. The angle β between the lines AB and BC is smaller than some threshold.

The first criterion enforces that some points on a long straight swipe are included. The second criterion is one that is more likely to be satisfied the further the three points are away from each other and the bigger the turn of the swipe is, as long as the turn is smaller than 90 degrees. If the turn is larger, then the third criterion is the one that includes the point.

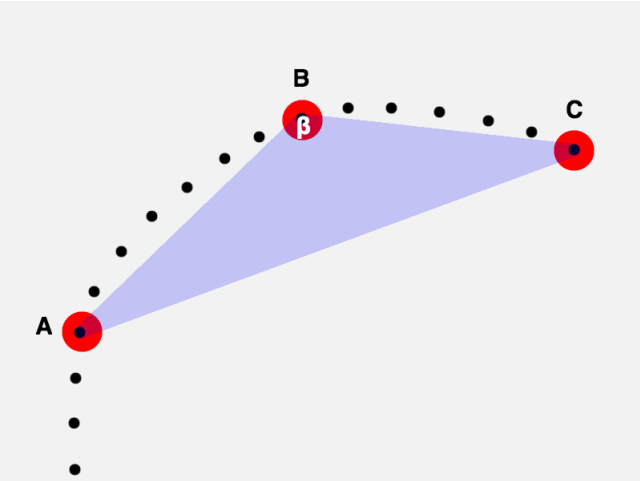

Figure 2.10: The small black points belong to the user's swipe reported by the operating system. The three red points are the last added point (A), the last point of the swipe (C) and a point that is in between the other two points (B). The area of the blue triangle and the angle β are two features that determine whether B is added to the input swipe.

2.4 Precalculation

So far, we have only considered how we can speed up our calculations after the user finished the swipe. In this section, it is described how it is possible to start some calculations while the user is still swiping. As we have seen in Section 2.2, for every potentially swiped word a table is build that simplifies finding the best mapping. Assume that we have such a table. Then, a new row can be added to the table for every new point in our swipe, i.e. we construct the tables row by row while the user is swiping. This is illustrated in Table 2.3 with the basic DTW algorithm for simplification.

2.5 Cache

In the last section we have seen that we can add rows to the table as the user swipes. When the user finished his swipe, the same thing can be done with the columns of the table. Assume that we built a table for the expected swipe of the word 'grand'. Then we can add new columns to that table and calculate the words 'grandfather' and 'grandmother' by calculating the table for the prefix 'grand' only once. In our implementation every prefix is cached and the calculations of a new word starts with the table of the longest prefix that is already calculated.

user swipe \ expected swipe	$(3,4)$	$(3,5)$	$(2,6)$	$(1,4)$	$(2,2)$
$(4,4)$	1	3	11	20	28
$(2,7)$	11	6	4	14	39
$(1,4)$	15	11	9	4	9
$(0,3)$	25	24	22	6	9
$(2,2)$	+5	+10	+16	+5	+0

Table 2.3: The point $(2,2)$ is added to the user's swipe. Then, the costs in the new row are calculated. Next, the costs will be added and the best mapping will have a score of 6.

2.6 Frequencies

There are a lot of words that have a similar swipe. For those words it is hard to decide which one should be displayed. If it is known which word is typed more often by the user, then the probability that this word was swiped is higher and will therefore be displayed. The frequencies of the words is integrated in the calculations as soon as the DTW returns the costs. Thus, it does not affect the cached or precalculated DTW tables.

Chapter 3

Implementation

The programming language used for writing the swipe keyboard is Swift. This language is interoperable with Objective-C, the language that is used for the Kännsch [6] keyboard on iOS, i.e. a predictive keyboard for flexible writing styles that occur in languages like swiss german. We integrated our swipe feature in the existing Kännsch keyboard. Some components of our swipe feature were adapted such that it works better in an environment that has less words, but a lot of them are user specific. In Chapter 4 we will investigate the runtime of the swipe feature using Kännsch and the runtime of our swipe keyboard using a larger dictionary.

3.1 Dictionary

The dictionary is the core element of our swipe keyboard. Only words that appear in the dictionary might be suggested for a swipe. Therefore, it is useful if the dictionary contains a lot of words. On the other hand, too much words in the dictionary reduces the quality of the suggested words. For the algorithm of Chapter 2 to work as fast as possible, an entry in the dictionary has to provide the following information:

- word
- frequency of the word
- expected length of the swipe
- start angle
- distance to the second letter
- point of the last letter
- perfect swipe

Since we are interested in a dictionary with a lot of entries, the memory consumption is crucial. The iOS keyboard extension has a significant lower memory limit than foreground apps [8] and therefore it is not possible to load 200′000 perfect swipes into memory. An approach to solve this is that the perfect swipe is not stored, but is calculated each time it is accessed. This is the case several hundred times per swipe as we will see in Chapter 4, but it saves a lot of memory. The memory consumption of the other properties are optimised by choosing the right data type. For example, the frequency is always an integer between 0 and 255 and therefore it can be stored as an unsigned byte instead of an integer, which takes up four bytes. With those optimisations the dictionary fits into the memory.

The data structure of the dictionary is optimised for the queries that run on it. For example, at the beginning of every swipe the start point filter (Section 2.1.1) is applied and the potential first letters of the swiped word are calculated. The words of the dictionary are stored in a two dimensional array of words. The first dimension denotes the first letter of the words stored in it. Therefore, we know all filtered words by calculating at most four indices of the potential first letters.

The dictionary is made persistent with a database. The words in the database with some of the properties listed above are loaded into memory each time the keyboard is opened.

3.2 Multithreading

In the keyboard there are several parts that run in parallel. As explained in the previous section, all the words are loaded into the memory as soon as the keyboard is opened. This takes some time and the user does not want to wait until all those words are loaded. Therefore, this is done in a background task with some synchronisation, because the loaded words might be needed for processing a swipe while the dictionary is still loading. In fact, all the calculations that are not directly dependent on the user interface are processed in some background task to keep the user interface responsive. Multithreading is also used is in the calculation of a swipe. There are a lot of words that need to be processed and the words do not affect each other during the calculation. Therefore, the swipe calculation can be sped up as evaluated in Chapter 4.

Chapter 4

Evaluation

This chapter evaluates the accuracy and the runtime of the different parts of our algorithm. The swipes that are used for this evaluation are collected on an iPhone 5 (997 swipes), an iPad 4th generation (749 swipes) and on an iPhone 6s (799 swipes). There are 1′248 words swiped in German and 1′297 words swiped in English. The German dictionary [9] that is used for the evaluation contains 205′828 words. The English dictionary [7] contains 160′606 words. If not stated otherwise, the German swipes are always compared to the full German dictionary and the English swipes are compared to the full English dictionary. The swipes for the evaluation are collected with an app that displays randomly drawn words and then the user has to swipe them. The probability that a word is drawn is the same as the probability that the word occurs in the corresponding language according to the dictionaries mentioned above.

4.1 Device

There are big differences in terms of accuracy between the iPhone 5, the iPhone 6s and the iPad as Figure 4.1 illustrates. The bigger the screen is, the better are the predictions. This is because the user's finger is not always very precise and therefore a small inexactness may lead to high costs on the iPhone 5, while the same error does not really matter on the iPad. The accuracy of the predicted word, i.e. the most likely word given a swipe, is about 5% lower than the accuracy of the suggested words, i.e. the four most likely words given a swipe. This is because some words have a similar expected swipe and it is hard to tell what the user intended to swipe. If we take the average accuracy of the three devices, then the probability that the predicted word is correct is 90.42% and the probability that the swiped word is suggested is 96.05%

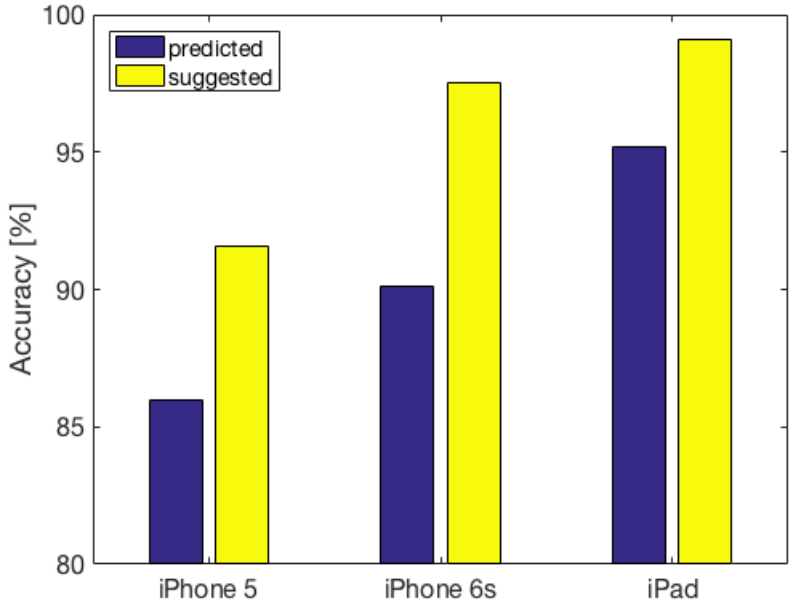

Figure 4.1: The effect of the device on the accuracy of our algorithm. The suggested words contain the four words that are most likely swiped. The word that is most likely swiped is called the predicted word.

4.2 Dictionary

Assume that the swiped word is in the dictionary. In Figure 4.2 we analysed how the number of words in the dictionary influences the accuracy. The accuracy of the suggested words gets worse as the dictionary's size increases. The accuracy of the predicted word stays around the 90% mark. If we would add more and more words to the dictionary, sooner or later the accuracy of the predicted word will decrease.

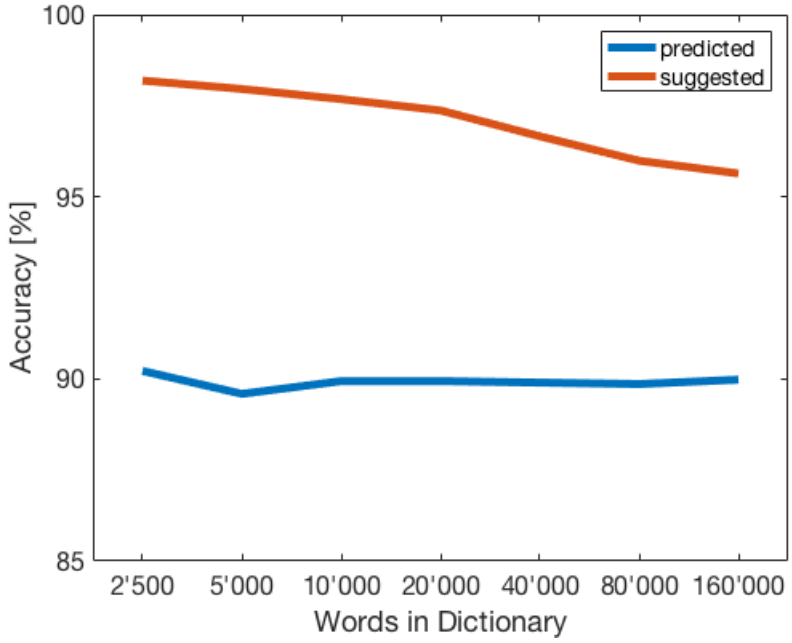

Figure 4.2: The effect of the number of words that are in the dictionary. The probability that the swiped word is contained in the suggested words, i.e. the four most likely swiped words, is between 95.64% and 98.19%. The probability that the predicted word is correct, i.e. the word that is most likely swiped, is between 89.59% and 90.22%.

4.3 Filters

This section evaluates the accuracy and runtime improvements of the filters discussed in Section 2.1. The parameters of the four filters are chosen in a conservative way, such that they do not influence the accuracy of the predictions too much. In Figure 4.3 it is shown that for each filter in over 99.45% of all cases the swiped word is included in the list of potential swiped words. The downside of being that conservative in choosing the parameters is the higher runtime of our algorithm. Depending on the dictionary's size, one could adjust the parameters such that real time performance is achieved. Even though the filters are chosen that conservative, most of the time only a few percents of the whole dictionary is filtered as Figure 4.4 shows. This means that the filters improve the runtime 264 times in the median case. The outliers are a problem, since they lead to a high runtime. One solution would be that we only take the $1'000$ most frequent words that are filtered. Another solution where we limit the runtime to 1 second will be discussed later in Section 4.6. Figure 4.5 shows how much words are filtered for each of the four filters. As we have seen earlier in Section 2.1.1, the start point

filter gives an upper bound of 22.95% for the English dictionary. The direction filter has some outliers at 100%. This is because the length of the user's swipe is shorter than the distance between two neighbouring letters on the keyboard and therefore the swipe ended before the direction was calculated. The outliers of the direction filter are no problem, because there are only a few words that have an expected swipe that short and hence the length filter works much better than in the usual case.

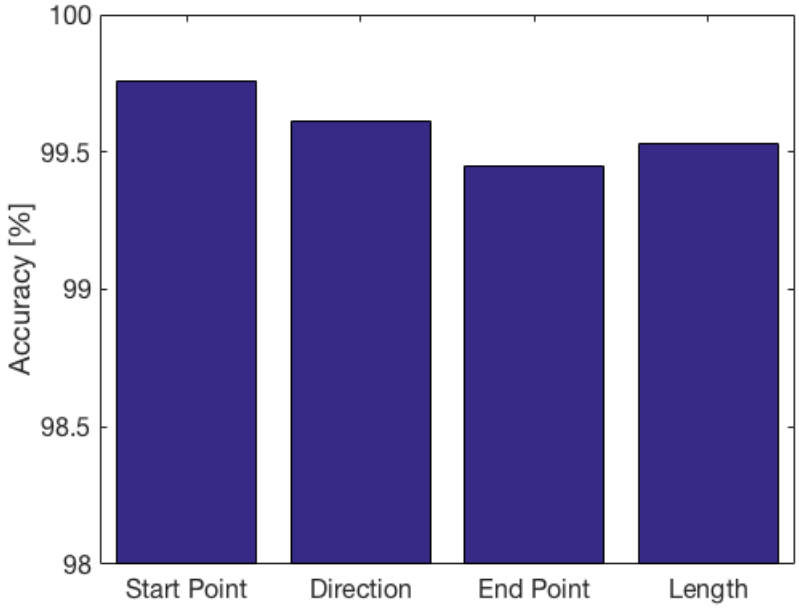

Figure 4.3: The accuracy of the filters indicate how often the swiped word was filtered, i.e. included in the list of potential swiped words.

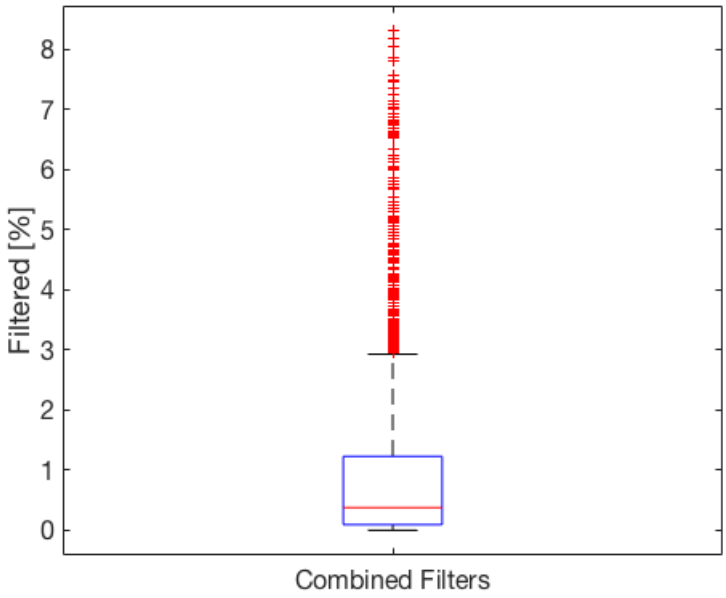

Figure 4.4: Percentage of words that are filtered, i.e. how big the list of the potential swiped words is compared to the full dictionary, using all four filters together.

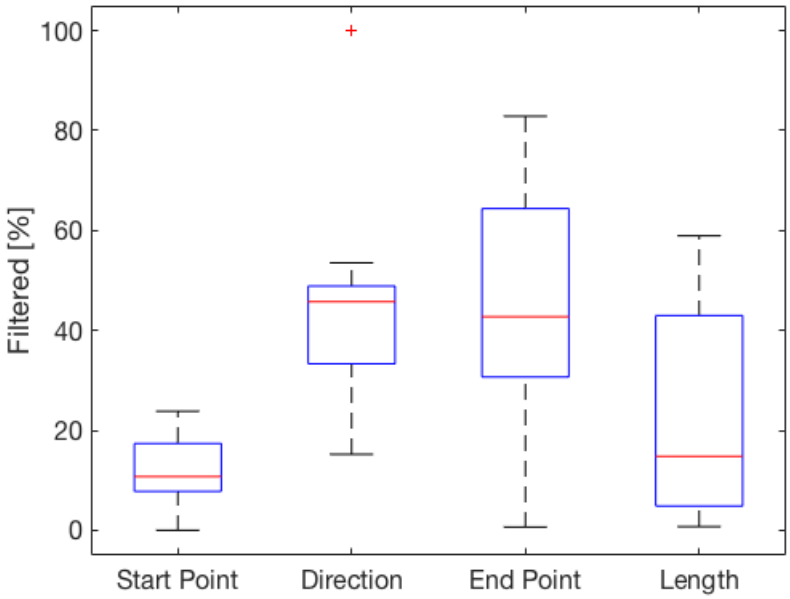

Figure 4.5: Percentage of the words that are contained in the list of potential swiped words separated by the four filtering techniques.

4.4 Input Swipe

In Section 2.3 we have seen a way to to reduce the number of points in the user's swipe. In Figure 4.6 the speedup of this method is analysed. Since the iPad has the biggest screen, the user swipes a longer time and therefore more points are in the user's swipe. That means that on an iPad more points can be ignored and hence a higher speedup is achieved than on the other devices. This method improves the accuracy of the predicted word from 86.92% to 89.98% and the accuracy of the suggested words from 92.97% to 95.64%.

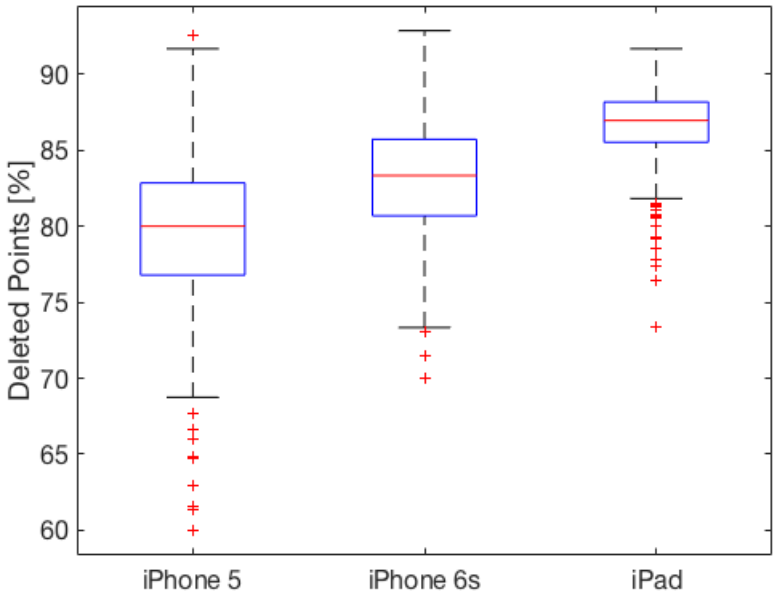

Figure 4.6: The amount of deleted points from the user's swipe depending on the device. Swipes on the iPad have a higher amount of deleted points and therefore our algorithm has a better speedup on this device.

4.5 Multithreading

Most of the current iOS devices have only 2 cores. Therefore, the achieved speedup is limited. For two cores the best achieved speedup is 1.41. This is because there is still some sequential and synchronised code required, e.g. for updating the list of suggested words. Theoretically, the speedup increases linearly with the number of cores until there are more cores than filtered words.

4.6 Runtime

In the last sections we have seen a lot of relative values. Now we will investigate how long a user of an iPhone 6s has to wait until the swiped word appears on the screen. In Figure 4.7 it is shown that the median runtime is 0.76 seconds. The longest runtime is 27.89 seconds. This is far too slow to be called realtime. This problem is solved by running the DTW for at most one second and cancel the calculations as soon as the time limit is reached. The accuracy of the correct words drops from 89.98% to 84.79% and the accuracy of the suggested words drops from 95.64% to 89.86%. This loss has to be accepted to guarantee real

time performance for large dictionaries with over 200′000 words. The Kännsch keyboard dictionary is a small one that contains the 5′000 most frequent words swiped by the user. In Figure 4.8 it is illustrated that the median runtime is 0.15 seconds and the maximal runtime is 0.71. Therefore, the time limit does not affect the accuracy.

Figure 4.7: Runtime of our algorithm using the full dictionary with 160′606 words (English) or 205′828 words (German).

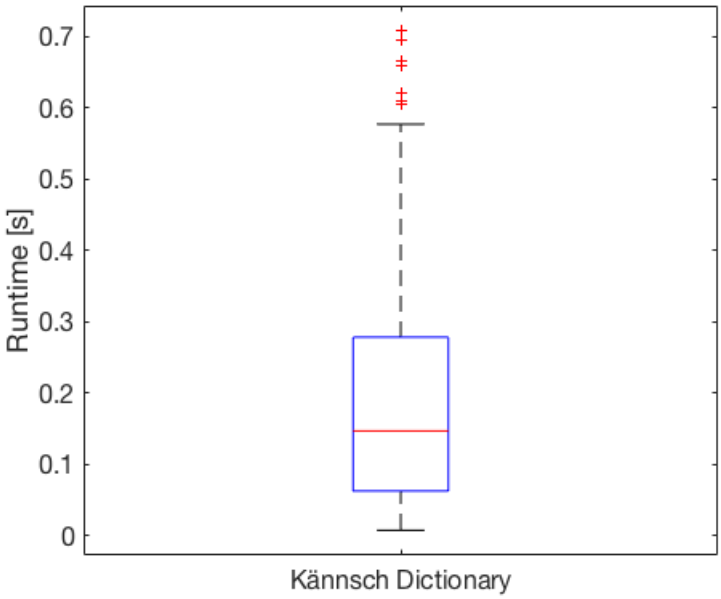

Figure 4.8: Runtime of our algorithm using the Kännsch dictionary that contains 5′000 words.

Chapter 5

Conclusion and Future Work

We developed a keyboard for iOS that supports swiping and we integrated this feature into an existing keyboard and published it on the App Store. We have chosen a deterministic approach that uses a dictionary and finds the words that matches best to the swipe. Further, several concepts were derived that improved the runtime. The most effective concept is filtering. With four filters, the dictionary was reduced 264 times in the median case. There are more concepts that can improve the performance. Therefore, a future work may find and implement more such concepts. The future work could also derive an algorithm that learns how a user swipes and adapts some parameters in order to improve performance and accuracy. In the Kännsch keyboard there is already a database that stores the frequencies of bigrams, i.e. the probabilities of the next words given the last one. This could be an interesting start point to improve the accuracy further, e.g. by varying the frequencies according to the last swiped word.

Bibliography

[1] Zhai, S., Kristensson, P.O.: The word-gesture keyboard: Reimagining keyboard interaction. Commun. ACM **55**(9) (September 2012) 91–101

[2] Smith, B.A., Bi, X., Zhai, S.: Optimizing touchscreen keyboards for gesture typing. In: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems. CHI '15, New York, NY, USA, ACM (2015) 3365–3374

[3] Nuance: Swype. http://www.swype.com Accessed: 2016-07-19.

[4] SwiftKey: SwiftKey. https://swiftkey.com Accessed: 2016-08-19.

[5] Google: Gboard. https://itunes.apple.com/us/app/gboard-search.-gifs.-emojis/id1091700242?mt=8 Accessed: 2016-08-19.

[6] Konrad, A.: Kännsch - Adaptive Keyboard for iOS. Bachelor's thesis, ETH Zurich (July 2016)

[7] Android: English wordlist. https://android.googlesource.com/platform/packages/inputmethods/LatinIME/+/master/dictionaries/en_US_wordlist.combined.gz Accessed: 2016-06-28.

[8] Apple: Creating an App Extension. https://developer.apple.com/library/ios/documentation/General/Conceptual/ExtensibilityPG/ExtensionCreation.html Accessed: 2016-06-21.

[9] Android: German wordlist. https://android.googlesource.com/platform/packages/inputmethods/LatinIME/+/master/dictionaries/de_wordlist.combined.gz Accessed: 2016-06-21.