

[Human-Keyboard-Interaction]: Final Report

[Lieyang Chen] 02/01, 2019

Professor: Janne Lindqvist Ph.D. Advisors: Gradeigh D. Clark
Human-Computer Interaction Lab Rutgers University, New Brunswick

Abstract

There are two main parts I did in this research. In the first part, I figured out how to use a C++ program to implement an algorithm which is used to generate all keyboard layouts that are needed. After the execution of this program, all xml files which are needed to change the AOSP keyboard layout will be generated. In the second part, I figured out how to use a Service object in Android Studio to make the AOSP keyboard layout change after a certain amount of time with those xml files generated in first part. Therefore, the modified AOSP keyboard has an extra feature that it can automatically change the keyboard layout periodically.

Introduction

The transition from one keyboard layout to a different keyboard layout needs to take several factors into consideration. First, we cannot directly change the keyboard layout into the final keyboard layout, because a sudden change of keyboard layout is not a good way for researching how users interact with a new keyboard layout. We have to make sure that users can smoothly get used to the new keyboard layout. A proper way to do this is to split keyboard transition into several stages, but how many stages do we need? The answer should be straightforward: it depends on how many keys are allowed to change in each stage. For example, if the original keyboard layout only differs from the new keyboard layout by three keys in total and we set that only 3 keys are allowed to change in one stage, then it is clear that we only need one stage in total to transition the keyboard layout.

However, we are not only pursuing to change three keys in one stage. We should allow researchers to customize the number of keys which are allowed to change in one stage. Also, we are not only pursuing one final keyboard layout. We should take any keyboard layouts into consideration. Moreover, we should allow researchers to customize the order of keys replacement. For example, a priority of the frequency of usage of keys may be adopted so that the most frequently used keys should be changed at first and any other keys should be taken care of later. Also, if researchers want to minimize the number of stages when being given the fixed number of keys that can be changed in one stage, they have to find the right order which allows such an optimization. (The process of finding such an order can be extremely hard. This will be clarified later) Therefore, it is important to design an algorithm which can easily handle any cases.

Algorithm1

The algorithm will be associated with several parameters, let's define them here:

int number : The fixed number of keys that need be changed in one stage. (It is also accepted if number-1 keys are get changed)

int stages : The number of stages associated with different n

string final[3] : The final keyboard layout (The string array should have a format like this: each string element represents a key row and each element should have a length of ten.

For example: f[0] = "qwertyuiop", f[1] = " asdfghjkl", f[2] = " zxcvbnm ")

string original[3] : The original keyboard layout (The string array should have a format like this: each string element represents a key row and each element should have a length of ten.

For example: f[0] = "qwertyuiop", f[1] = " asdfghjkl", f[2] = " zxcvbnm ")

String order : The order of keys replacement. (This string should have a length of 26 and all characters are unique. For example: order = "qwertyuiopasdfghjklzxcvbnm")

key_pos pos_array[number] : used to store the position of each mismatched key pair (key_pos is a customized data type which has two attributes: int row, int col, col stands for column)

char key_array[number+1] : used to store the mismatched keys

The algorithm should take three elements as inputs: final, number and order. (We assume the algorithm has a default original = {"qwertyuiop", " asdfghjkl", " zxcvbnm "). The algorithm will call original.transition(final,number,order) first.

Pseudo:

```
transition(final,number,order)
    for i = 0 .. 25
        pos = position of order[i] in original;
        if (original[pos.row][pos.col] != final[pos.row][pos.col])
            key1 = original[pos(i).row][pos(i).col];
            counter = 0;
            replacement(number,counter,key1,final,counter);
            file_create(); // every time replacement(..) returns,
one stage will be generated, so the algorithm will generate the
corresponding xml files according to the current layout.
```

```
replacement(number,counter,key1,final,counter)
    key_array[0] = key1;
    for i = 0..number-1
        pos_array[i] = position of key_array[i]
        key_array[i+1] = final[pos_array[i].row][pos_array[i].col]
        counter++;
        if (key_array[i+1]==key1)
            break;

    for i = counter..1
        swap(pos_array[counter],pos_array[i-1])
```

```

number2 = number - counter

if (number2 > 1)
    find_mismatch(final,number2,order)

find_mismatch(final,number2,order)
    for i = 0 .. 25
        pos = position of order[i] in original;
        if (original[pos.row][pos.col] != final[pos.row][pos.col])
            key1 = original[pos(i).row][pos(i).col];
            counter = 0;
            replacement(number,counter,key1,final,counter);

swap(pos1,pos2)
    temp_key = original[pos1.row][pos1.col];
    original[pos1.row][pos1.col]=original[pos2.row][pos2.col];
    original[pos2.row][pos2.col]=temp_key;

file_create()
    according to current layout, all corresponding xml files will be
    generated inside the designated directory.

```

To more specifically explain this process, here I give an example:

```

Let order = "qwertyuiopasdfghjklzxcvbnm"
number = 6
final = "qwdrtuylkpzasehniomxfcvgbj"
key_pos pos_array[6];
char key_array[7];
original = "qwertyuiopasdfghjklzxcvbnm"

```

Stage 1:

First iteration:

According to order, the algorithm will find the first mismatch keys at 'e' in original and at 'd' in final. It will push 'e' onto key_array first and push 'd' on to key_array next. It will use 'e' to find the position of 'e' in original, which is (0,2). It will push (0,2) onto pos_array.
key_array = ['e','d']
pos_array = [(0,2)]

Second Iteration:

Find the position of 'd' in `original`, which is (1,3), and push it onto `pos_array`. Use (1,3) to find the key in `final` which is 's'. Push 's' onto `key_array`.

```
key_array = ['e','d','s']
```

```
pos_array = [(0,2),(1,3)]
```

Third Iteration:

Find the position of 's' in `original`, which is (1,2), and push it onto `pos_array`. Use (1,2) to find the key in `final` which is 'a'. Push 'a' onto `key_array`.

```
key_array = ['e','d','s','a']
```

```
pos_array = [(0,2),(1,3),(1,2)]
```

Forth Iteration:

Find the position of 'a' in `original`, which is (1,1), and push it onto `pos_array`. Use (1,1) to find the key in `final` which is 'z'. Push 'z' onto `key_array`.

```
key_array = ['e','d','s','a','z']
```

```
pos_array = [(0,2),(1,3),(1,2),(1,1)]
```

Fifth Iteration:

Find the position of 'z' in `original`, which is (2,2), and push it onto `pos_array`. Use (2,2) to find the key in `final` which is 'x'. Push 'x' onto `key_array`.

```
key_array = ['e','d','s','a','z','x']
```

```
pos_array = [(0,2),(1,3),(1,2),(1,1),(2,2)]
```

Sixth Iteration:

Find the position of 'x' in `original`, which is (2,3), and push it onto `pos_array`. Use (2,3) to find the key in `final` which is 'f'. Push 'f' onto `key_array`.

```
key_array = ['e','d','s','a','z','x','f']
```

```
pos_array = [(0,2),(1,3),(1,2),(1,1),(2,2),(2,3)]
```

Through six iterations, even though 7 keys are collected, we will only need to use previous 6 keys actually. We now iteratively performing the swapping with `pos_array` elements by 5 times:

First swapping:

```
original[2,3] <-> original[2,2] : x <-> z
```

Second swapping:

```
original[2,3] <-> original[1,1] : z <-> a
```

Third swapping:

```
original[2,3] <-> original[1,2] : a <-> s
```

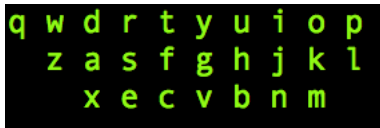
Forth swapping:

```
original[2,3] <-> original[1,3] : s <-> d
```

Fifth swapping:

```
original[2,3] <-> original[0,2] : d <-> e
```

Therefore, the `current_layout` will be like:



```
q w d r t y u i o p
z a s f g h j k l
x e c v b n m
```

After this, the algorithm is going to continue with order.

Stage 2:

First iteration:

The algorithm will find the first mismatch keys at 'y' in `current_layout` and at 'u' in `final`. It will push 'y' onto `key_array` first and push 'u' onto `key_array` next. It will use 'y' to find the position of 'y' in `current_layout`, which is (0,5). It will push (0,5) onto `pos_array`. (note: the `key_array` and `pos_array` are recreated here)

`key_array = ['y','u']`

`pos_array = [(0,5)]`

Second Iteration:

Find the position of 'u' in `current_layout`, which is (0,6), and push it onto `pos_array`. Use (0,6) to find the key in `final` which is 'y'. Push 'y' onto `key_array`.

`key_array = ['y','u','y']`

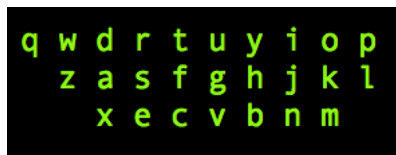
`pos_array = [(0,5),(0,6)]`

Here, the algorithm will find that the newly found key 'k' is same as the first key in the `key_array`. Then it will end the rest of iterations and perform the swapping right away. Since two positions are collected, the algorithm only needs to perform swapping once.

First swapping:

`current_layout[0,6] <-> original[0,5] : u <-> y`

Therefore, the `current_layout` will be like:



```
q w d r t u y i o p
z a s f g h j k l
x e c v b n m
```

Now the algorithm only swapped two keys, it still has four keys left to be swapped. Therefore, it will restart checking mismatch on `current_layout` using order:

First iteration:

The algorithm will find the first mismatch keys at 'i' in `current_layout` and at 'l' in `final`. It will push 'i' onto `key_array` first and push 'l' onto `key_array` next. It will use 'i' to find the

position of 'i' in `current_layout`, which is (0,7). It will push (0,7) onto `pos_array`. (note: the `key_array` and `pos_array` are recreated here)

```
key_array = ['i','l']
```

```
pos_array = [(0,7)]
```

Second Iteration:

Find the position of 'l' in `current_layout`, which is (1,9), and push it onto `pos_array`. Use (1,9) to find the key in `final` which is 'm'. Push 'm' onto `key_array`.

```
key_array = ['i','l','m']
```

```
pos_array = [(0,7),(1,9)]
```

Third Iteration:

Find the position of 'm' in `current_layout`, which is (2,8), and push it onto `pos_array`. Use (2,8) to find the key in `final` which is 'j'. Push 'j' onto `key_array`.

```
key_array = ['i','l','m','j']
```

```
pos_array = [(0,7),(1,9),(2,8)]
```

Forth Iteration:

Find the position of 'j' in `current_layout`, which is (1,7), and push it onto `pos_array`. Use (1,7) to find the key in `final` which is 'i'. Push 'i' onto `key_array`.

```
key_array = ['i','l','m','j','i']
```

```
pos_array = [(0,7),(1,9),(2,8),(1,7)]
```

Again, the algorithm will find that the newly found key 'i' is same as the first key in the `key_array`. Then it will end the rest of iterations and perform the swapping right away. Since four positions are collected, the algorithm only needs to perform swapping three times.

First swapping:

```
current_layout[1,7] <-> original[2,8] : j <-> m
```

Second swapping:

```
current_layout[1,7] <-> original[1,9] : m <-> l
```

Third swapping:

```
current_layout[1,7] <-> original[0,7] : l <-> i
```

Therefore, the `current_layout` will be like:

```
q w d r t u y l o p
  z a s f g h i k m
    x e c v b n j
```

Now the algorithm has swapped 6 keys, it has finished one stage. Therefore, the algorithm is going to continue with order.

Stage 3:

First iteration:

The algorithm will find the first mismatch keys at 'o' in `current_layout` and at 'k' in `final`. It will push 'o' onto `key_array` first and push 'k' onto `key_array` next. It will use 'o' to find the position of 'o' in `current_layout`, which is (0,8). It will push (0,8) onto `pos_array`. (note: the `key_array` and `pos_array` are recreated here)

```
key_array = ['o','k']
```

```
pos_array = [(0,8)]
```

Second Iteration:

Find the position of 'k' in `current_layout`, which is (1,8), and push it onto `pos_array`. Use (1,8) to find the key in `final` which is 'o'. Push 'o' onto `key_array`.

```
key_array = ['o','k','o']
```

```
pos_array = [(0,8),(1,8)]
```

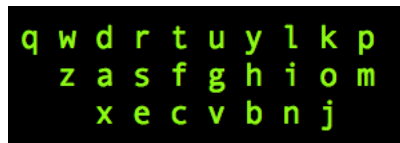
Here, the algorithm will find that the newly found key 'o' is same as the first key in the `key_array`. Then it will end the rest of iterations and perform the swapping right away.

Since two positions are collected, the algorithm only needs to perform swapping once.

First swapping:

```
current_layout[1,8] <-> original[0,8] : k <-> o
```

Therefore, the `current_layout` will be like:



Now the algorithm only swapped two keys, it still has four keys left to be swapped. Therefore, it will restart checking mismatch on `current_layout` using order :

First iteration:

The algorithm will find the first mismatch keys at 'f' in `current_layout` and at 'e' in `final`. It will push 'f' onto `key_array` first and push 'e' onto `key_array` next. It will use 'f' to find the position of 'f' in `current_layout`, which is (1,4). It will push (1,4) onto `pos_array`. (note: the `key_array` and `pos_array` are recreated here)

```
key_array = ['f','e']
```

```
pos_array = [(1,4)]
```

Second Iteration:

Find the position of 'e' in `current_layout`, which is (2,3), and push it onto `pos_array`. Use (2,3) to find the key in `final` which is 'f'. Push 'f' onto `key_array`.

```
key_array = ['f','e','f']  
pos_array = [(1,4),(2,3)]
```

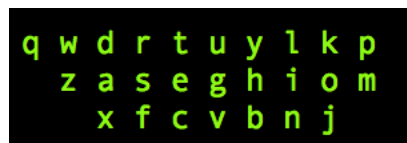
Here, the algorithm will find that the newly found key 'f' is same as the first key in the key_array. Then it will end the rest of iterations and perform the swapping right away.

Since two positions are collected, the algorithm only needs to perform swapping once.

First swapping:

```
current_layout[2,3] <-> original[1,4] : e <-> f
```

Therefore, the current_layout will be like:



```
q w d r t u y l k p  
z a s e g h i o m  
x f c v b n j
```

Now the algorithm only swapped two keys, it still has two keys left to be swapped. Therefore, it will restart checking mismatch on current_layout using order:

First iteration:

The algorithm will find the first mismatch keys at 'g' in current_layout and at 'h' in final. It will push 'g' onto key_array first and push 'h' onto key_array next. It will use 'g' to find the position of 'g' in current_layout, which is (1,5). It will push (1,5) onto pos_array. (note: the key_array and pos_array are recreated here)

```
key_array = ['g','h']
```

```
pos_array = [(1,5)]
```

Second Iteration:

Find the position of 'h' in current_layout, which is (1,6), and push it onto pos_array. Use (1,6) to find the key in final which is 'n'. Push 'n' onto key_array.

```
key_array = ['g','h','n']
```

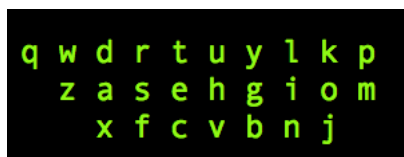
```
pos_array = [(1,5),(1,6)]
```

Through two iterations, even though 3 keys are collected, we will only need to use previous 2 keys actually. We now iteratively performing the swapping with pos_array elements by 1 times:

First swapping:

```
original[1,6] <-> original[1,5] : h <-> g
```

Therefore, the current_layout will be like:



```
q w d r t u y l k p  
z a s e h g i o m  
x f c v b n j
```


Now the algorithm has swapped 6 keys, it has finished one stage. Therefore, the algorithm is going to continue with order.

Stage 4:

First iteration:

The algorithm will find the first mismatch keys at 'g' in `current_layout` and at 'n' in `final`. It will push 'g' onto `key_array` first and push 'n' onto `key_array` next. It will use 'g' to find the position of 'g' in `current_layout`, which is (1,6). It will push (1,6) onto `pos_array`. (note: the `key_array` and `pos_array` are recreated here)

```
key_array = ['g','n']
```

```
pos_array = [(1,6)]
```

Second Iteration:

Find the position of 'n' in `current_layout`, which is (2,7), and push it onto `pos_array`. Use (2,7) to find the key in `final` which is 'b'. Push 'b' onto `key_array`.

```
key_array = ['g','n','b']
```

```
pos_array = [(1,6),(2,7)]
```

Third Iteration:

Find the position of 'b' in `current_layout`, which is (2,6), and push it onto `pos_array`. Use (2,6) to find the key in `final` which is 'g'. Push 'g' onto `key_array`.

```
key_array = ['g','n','b','g']
```

```
pos_array = [(1,6),(2,7),(2,6)]
```

Here, the algorithm will find that the newly found key 'g' is same as the first key in the `key_array`. Then it will end the rest of iterations and perform the swapping right away.

Since three positions are collected, the algorithm only needs to perform swapping twice.

First swapping:

```
current_layout[2,6] <-> original[2,7] : b <-> n
```

Second swapping:

```
current_layout[2,6] <-> original[1,6] : n <-> g
```

Therefore, the `current_layout` will be like:

```
q w d r t u y l k p
  z a s e h n i o m
    x f c v g b j
```

Here, even though we still have three keys left to be swapped, the `current_layout` is already same as `final`. Therefore, the algorithm will find no mismatch and return right away.

Complexity: because this algorithm takes variables with limited size, so the running time of this algorithm is also constant.

Well, now we have an algorithm that can provide researchers with all stages of layouts given specified `final`, `number` and `order`. However, what if researchers want to minimize the number of stages when providing the specific `final` and `number`. In this situation, we may need to find which orders can make this optimization. Therefore, we may need an algorithm2 to generate all possible orders, use algorithm1 to test all of them and put all orders that satisfy the optimization in a set.

Algorithm2

The clarification of the algorithm two is included inside `kbd_combination.txt` file (in the `kbd_combination_C++` folder).

However, this algorithm is really limited. Because we have an order which has a length of 26, the total number of possible orders that can be generated is the factorial of 26. Moreover, since we cannot skip any possible orders, we have to make sure we need to go through all of them. This will take extremely long time for a computer to compute. Therefore, the algorithm can only work if the length of order is small, but 26 is so large for it that the problem has to be left unsolved.

Complexity: $O(N!)$

Automatic Switching of Keyboard Layout

Now we are able to customize the transition of keyboard, so all required xml files to switch keyboard layout on an android device are generated. The next goal is to design a way that makes the keyboard layout on participants' Android device automatically change without any manual help from researchers. Namely, we want the keyboard layout on the users' Android device to automatically change after a certain amount of time.

Process:

We first put all xml files inside the globally accessible array and create a globally accessible counter. Then we create a Service object inside AOSP. This object will trigger an alarm inside Android device so that, after a certain amount of time, the counter will be incremented by one

and the element specified by the counter in xml file array will be used to create the keyboard layout.

Future Goals

By adding more features into the AOSP keyboard, we have a keyboard application that can allow users to experience different types of keyboard layouts. Because our final goal is to figure out which keyboard layout will be outstanding from others, we will need to record useful parameters like the number of words typed by users per minute, typing speed, error rate and so on. We will need to figure out how to store these useful information in local device using SQLite as well as remote database like Google Firebase.

Conclusion

The research experience is really wonderful, I learned how to design an algorithm on my own when facing some real-world problem, learned a lot of APIs used in android studio and, most importantly, learned how to be self-motivated and how to study a completely new field on my own. At last, I want to appreciate Professor Lindqvist and my Advisor. Thanks for you to give me an opportunity to have a wonderful experience of academic research. Thank you very much!