A Genetic Algorithm for Finding the Optimal Keyboard for Typing Speed

Steven Abreu, Dorian Levine, Marouane Abra, Toby Harvey

1. INTRODUCTION

Keyboards throughout history have not just been created to achieve faster and faster possible typing speeds. On the contrary, some keyboard layouts are designed to slow down users. This is actually the case for QWERTY, which most of use. QWERTY was created so that users would not jam typewrites, which was pretty easy to do given their physical limitations. But now that we are typing on digital computers, with slick keyboards, there is no reason to try to avoid the fastest possible keyboard. The Dvorak Keyboard is an example of a keyboard designed for easier use. Our project was to try to find a Keyboard that is easier to use than QWERTY ourselves.

2. RELATED WORK

We found two other projects similar to ours. *In Evolving a More Optimal Keyboard*, a somewhat similar fitness function to ours is used, where the keyboard is penalized for having to use same finger/hand multiple times, and fingers are assessed by strength (see next section). Michael Capewell has a website, that links code for his genetic algorithm on keyboard layouts, but has not explanation or literature along with it so it was largely ignored. For the most part all of our work we came up with ourselves.

3. PROBLEM EXPOSITION

In order to find a good keyboard for fast typing speeds, we have 32! possible keyboards to choose from. (select a position for 1 letter, you have 31 options for the next, and 30 for the next after that...). 32! is a pretty larger search space, so we did not just use a brute for approach, but instead a genetic algorithm. We can divide our genetic algorithm into a few sections. 1. Our fitness function (How do we simulate typing?) 2. Our breed and mutate functions. 3. Our generation to generation selection.

By using a genetic algorithm our goal was to be able to traverse the search space to a minimum (the lower fitness the better) faster, assuming our breed function in fact produced on average better offspring, and our fitness function does a good job simulating what keyboard is easier to type on.

3.1 Fitness Function

To assess the fitness of a keyboard, we attempted to simulate how difficult it would be to type a given amount of text on a given Keyboard. To this end, we read a text file in a single character at a type, and increment the fitness of the given keyboard a character at a time. Four factors go into how much we increment the fitness per character:

—The strength of the finger used to type.

—Whether or not that finger has been used repeatedly.

—Whether or not that hand has been used repeatedly.

—The distance that finger must move in order to type the given character.

Diagram.jpg

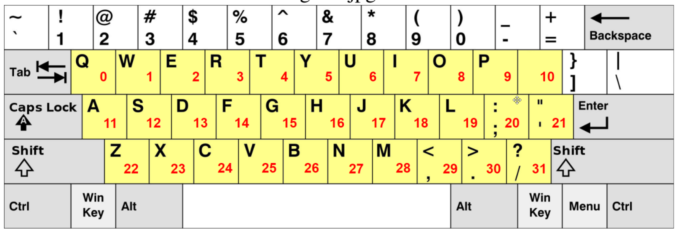
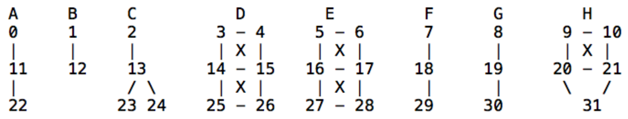


diagram.png



3.1.1 *Strength of Finger.*

The strength of each finger is simply set to a constant, and then every time that finger is used to type a character that strength is added to the fitness. In this case the lower "strength" the better. Where from left pinky to right pinky the array looks like [15,7,5,5,5,5,7,15].

3.1.2 *Finger repetition.*

A Finger repetition penalty is given every time a finger has to be used multiple times in a row. The penalty is set so that it increments by 1 for the first 4 consecutive touches, and after that increments by number of consecutive touches divded by 2.

3.1.3 *Hand repetition.*

A Hand repetition penalty is given every time a Hand has to be used multiple times in a row. This penalty is set at a constant. Every time that a hand is used consecutively we add 3 to the fitness.

3.1.4 *Distances.*

The distances to a key are a more complicated matter. As seen in the above diagram we create Graphs A through H which are associated with each finger from left pinky to right pinky, respectively. Each node of these graphs are associated with a key on the keyboard, and each edge of the graph indicates that the associated finger can move between the two keys or nodes. For graphs A, B, C, F, and G the distance between keys is stored in the graph representation and added to the total fitness when that key is pressed. The finger that pressed that key immediately traverses back to the starting position key for that graph. For instance on QWERTY if we need to press Q we move the pinky from A to Q add the distance and then move directly back to A. For graphs D and E we add a 4 character delay so that if we press a key away from the starting position our index finger stays on that key for 3 iterations of characters. If we must press another character on that same finger before 4 iterations, we run Dijkstra's algorithm to to find the shortest path to the new key, and begin the delay again. This tactic is implemented because we believe that it is unrealistic for the typer to not optimize by attempting to traverse keys used by index fingers in a shortest path manner. No one would type a "T" and then return to "F" if they knew that hand to immediately type a ”G” after the ”T”. Lastly we run Dijkstra’s algorithm on graph H but with no delay.

3.2 Breed and Mutate Functions

A mutation of a keyboard is simply a switch of two random keys in that keyboard to create the new one. This operation must be simple in order to try to generate a new keyboard whilst still keeping the low fitness of the parent keyboard. Only switching one pair of keys guarantees the new keyboard will be very close to the previous in terms of fitness, and hopefully it will improve just a little bit.

The breeding of two keyboards was something our group really had to think about to figure out the best implementation. We settled on taking every other graph of parent 1 and adding it to the child keyboard, and then filling the rest of the child keyboard with the graphs from parent 2.

In detail: Add graphs A, C, E and G of parent 1 to child keyboard Add graphs B, D, F and H of parent 2 to child keyboard Fill replace duplicate keys with random characters that have not been used yet.

This algorithm pretty effectively breeds two keyboards together. Much of the integrity of the two parent keyboards still exists (the graphs, shortest paths, key combinations possible, etc) however the child keyboard is completely different from both parents. This can have both a positive effect or negative effect. However when run enough times, the trend of keyboard fitness was improving, so it must have worked at least a little.

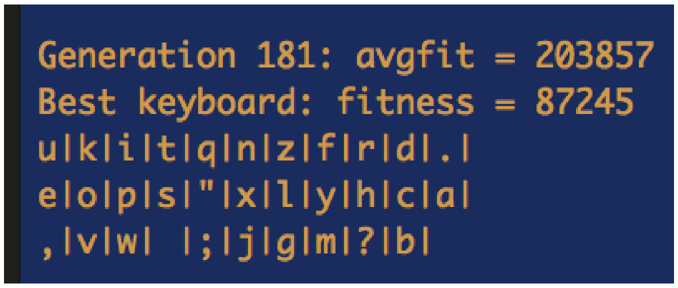
3.3 Overarching Genetic and Generation to Generation Algorithm

The genetic algorithm integrates the fitness, breed, and mutate functions to generate groups of increasingly efficient keyboard layouts. Initially, the algorithm creates a pool of 100 randomly generated keyboard layouts. From there, each keyboard is fed a string of text that is used to compute its fitness level. The average fitness level of all 100 keyboards is stored for later use. At this point, the selection and breeding process begins. Two parents are randomly chosen from the pool and are assigned a breeding probability based on their fitness relative to the average fitness. The equation to calculate the breeding probability is:
$(\frac{avgFitness}{fitness(parent1)}) \cdot (\frac{avgFitness}{fitness(parent2)})$

Thus, the chance that two parents will breed increases proportionally based on how much more efficient they are than the average. This means that more efficient groups of parents will breed more often, and less efficient groups of parents will breed less often, leading to a trend of increasing efficiency as more generations are created. If two parents successfully breed, the offspring is then run through the mutate function and added to the next generation of keyboards. The breeding and selection process is repeated until 100 new offspring are produced. Once this condition is satisfied, the fitness is calculated for each new keyboard and the breeding and selection process begins again from there, continuing until a user-defined number of generations are bred.

4. FINDINGS AND RESULTS

We have ran our algorithm on 5 test files of text. Although we would like to run in on significantly larger pieces of text, we do in fact see a drop in fitness so that we know the algorithm is at least minimizing our fitness function a bit. What we would ultimately like to see is keyboards where common letters, and strings of letters are easy to type for a given document. In the above figure is a keyboard that scored a pretty low fitness on the first paragraph of a Harry Potter book. We are not completely satisfied with our results, and recognize that what our keyboard looks like is largely due to our breed and fitness functions which rely on understanding how typing actually works, something that we are not completely sure we understand that well. There is a sort of paradox here, we are building a system to give us a better keyboard, but to understand how to build that keyboard we would already in someways need to know what the best keyboard is. In this case this is only a small attempt and we could probably continue trying to model typing and what a good keyboard is for ever.



```
Generation 181: avgfit = 203857
Best keyboard: fitness = 87245
u|k|i|t|q|n|z|f|r|d|.|
e|o|p|s|"|x|l|y|h|c|a|
,|v|w| |;|j|g|m|?|b|
```

REFERENCES

Christopher P Walker, *Evolving a More Optimal Keyboard,* http://web.mst.edu/ tauritzd/courses/ec/fs2003/project/Walker.pdf, (2003).

Michael Capewell, *Keyboard Evolve,* http://www.michaelcapewell.com/programming/keyboardevolve.htm, (2005).