Computer Science EE:

To what extent can Genetic Algorithms optimize the English Keyboard Layout for Speed?

Session: May 2018

Word Count: 3985

Table of Contents

Introduction

We use the QWERTY keyboard every day but is it really the best layout to use? This paper tries to find a better layout by using a modern optimization technique, treating the question: To what extent can Genetic Algorithms optimize the English Keyboard Layout for Speed? First, it covers metaheuristic algorithms in general, poses the problem of finding an ideal keyboard layout and the process of solving it effectively with a Genetic Algorithm (GA), and then it critically evaluates the solution. To solve this optimization problem multiple, scholarly sources on optimization and GAs were examined. From this knowledge, I wrote a Java program that implements a GA from scratch to find a solution to the keyboard layout problem.

The purpose of this research is to analyze and find value in 'soft-computing' techniques for solving everyday optimization problems. Soft-computing -- the process of approximating solutions dynamically instead of hard-coding known algorithms -- is a field that consists of nature-inspired algorithms to solve difficult problems, and can create working, low-cost solutions in little amounts of time (Korošec). More specifically GAs model Darwin's Theory of Evolution: by leveraging 'survival of the fittest,' genetic mutations, and reproduction, they can traverse a large search space of solutions to find a near-optimal solution to a problem. I researched scientific and reputable papers and found that the scope of GAs is extensive, as they can be used to solve a range of problems in many fields. Going through the process of solving my own problem, lead to the conclusion that GAs are useful, easy to implement and good for solving optimization problems but not ideal for every scenario: they can optimize the keyboard to a fair extent. [278 words]

Metaheuristic Algorithms

Metaheuristic algorithms are efficient in solving optimization problems in a reasonable amount of time using a form of trial and error. In mathematics and computer science, an optimization problem is one in which *the* best or optimal solution must be found in a finite or infinite set of solutions (Korošec). Optimization problems in which the search space is unfeasibly large could potentially take years to solve with brute force methods (methods in which each element of a potential solution set is evaluated one after another). As such, metaheuristic algorithms offer a better approach to finding acceptable solutions by making approximations (soft-computing) rather than hard-computing with predetermined algorithms. The solutions to metaheuristic algorithms, however, are usually not guaranteed optimal: in fact, there is no way of knowing whether a metaheuristic algorithm produces the best solution, or even whether it will work and why if it does work -- naturally raising skepticism. Nonetheless, solutions will be *good* (Gandomi).

Common nature-inspired algorithms include artificial neural networks, machine learning, fuzzy logic, swarm-intelligence-based algorithms, and more importantly for this paper, GAs (Gandomi). GAs find applications in many fields, especially in biomedicine and engineering: they are useful for designing *things.* Say you wanted to make a turbine blade. One method is to hire an engineer who provides exact blueprints based on mathematical analysis to come up with a solution: this would be comparable to implementing a specific, known algorithm. However, in design problems with too many variables (width, height, radius, shape, etc.) it is hard for the engineer to come to a definite analytical solution reliably. Instead, using a GA, evolution can find a blade design that dominates the rest by simulating every variable and letting it find the fittest over many generations. The algorithm does not require knowledge of *how* the optimal blade is obtained, it just produces it after many generations of evolution, starting from randomly simulated blades and ending at near-optimal ones. In this paper, the main ideas of GAs will be examined and will be demonstrated with an interesting, everyday optimization problem: finding the fastest keyboard layout.

Genetic Algorithms

Nature is surprisingly smart: by creating organisms based on their DNA, letting the fittest organisms reproduce to create new DNA, and by providing means of genetic exchange and mutation, evolution over generations becomes one of the forerunners in metaheuristic optimization problem-solving algorithms (Malhotra). As such, there is a branch of computer science dedicated to the study and development of Evolutionary Algorithms (EAs) -- algorithms inspired by evolution to solve optimization problems. One popular variant of EAs is the GA which is useful for effectively traversing a large search space and finding a good answer to a complex problem. In general, GAs work by modeling a population of species, where each organism has its own genome, and by letting the population evolve through the principles of natural selection, reproduction, and mutation (these concepts will be explained in more detail later).

In GAs, a genome is usually modeled by an array of bits, words, or objects. Based on its genome, the individual's fitness can be calculated and represented in some way (Levin). For example, the fitness of an individual who is supposed to model a 'goal' sequence of letters "HELLO," may have a genome sequence "HKLRO". In this case, a good fitness function would count the letters of the genome that correspond to the letters of the goal sequence, so an individual above would have a fitness of 2 since the letters 'H' and 'O' are in the right place. Upon evaluating the fitness for each individual in the population, the algorithm 'kills' the least fittest individuals and replaces them with 'offspring' of the fittest individuals, so that traits of the fittest individuals get passed on, slowly populating new generations, and ultimately finding an optimal or near-optimal solution (Levin).

The Keyboard Optimization Problem

One may think the keyboard layout we use every day is efficient. Surprisingly though, the standard US QWERTY keyboard is not an optimal layout for speed typing. In fact, it is very sub-optimal, since frequently typed letters are placed on opposite ends to slow down typists and historically, keep typewriters from jamming (Hempstalk). So naturally, the question arises: how can we find layouts that are faster than QWERTY?

This paper focuses on finding the fastest, theoretical keyboard layout in the set of all possible layouts, so it is an optimization problem. Popular statistically derived solutions (based on English letter frequency) include speed typing keyboards such as the Dvorak and the Colemak layout which were developed by researchers to maximize speed (Hempstalk).

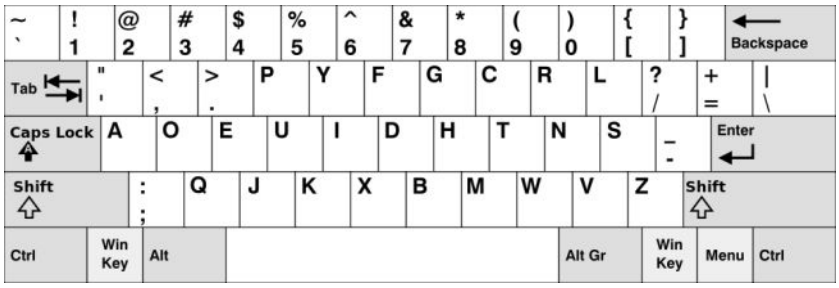


Fig. 1. Colemak Layout

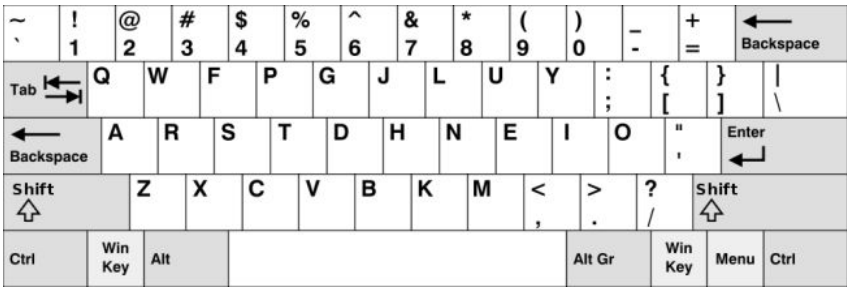


Fig. 2. Dvorak Layout

Instead of mathematical analysis, this paper aims to find optimal solutions that may compete with Dvorak and Colemak keyboards using a GA. This method is more abstract as it relies on a simulation of nature rather than mathematics. However, it is also where the benefits of soft-computing shine: a program can optimize a problem without relying on the programmer's knowledge. It finds a near-optimal solution without needing the exact steps to derive a solution.

Evaluating Fitness

Now that the problem is familiar we must consider the first step to solving an optimization problem: representing a general solution. In the case of the keyboard problem, a keyboard layout (that represents the main three key-rows) can be represented by a 33-dimensional vector, where each component represents a letter of the keyboard. This vector is visualized as follows.

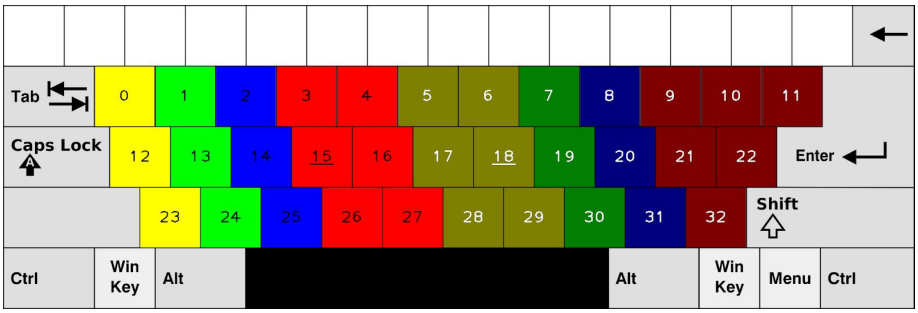
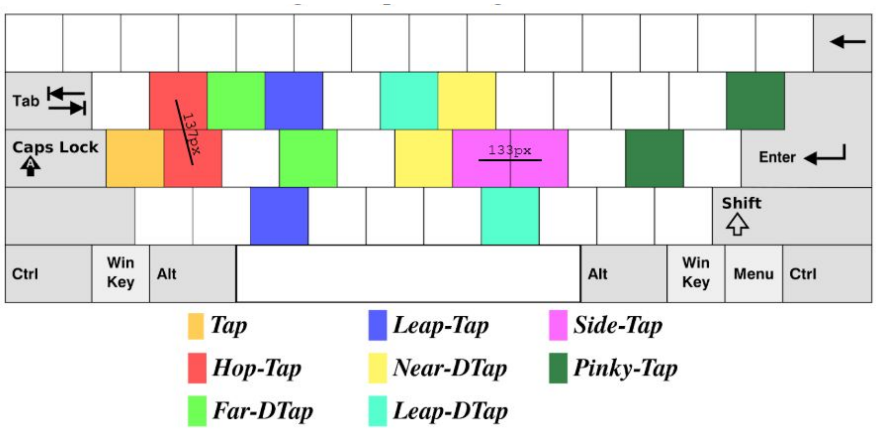


Fig. 3. General keyboard layout. The numbers correspond to the components of the vector and each color corresponds to a finger (light-yellow: left-pinky, light-green: left-ring, etc). This keyboard object is *List* of 33 *'Key'* objects in Java.

The second step is to construct an *objective* (also known as a *cost* function), that maps a vector to a scalar output, representing the cost of that specific vector. In the context of GAs, this function is referred to as the *fitness function* and determines how fit an individual is (Levin). The fitness function finds the efficiency of a given keyboard layout vector by finding the theoretical time it takes to type a given text.

Inspired by Malas et al. *Toward optimal Arabic keyboard layout using genetic algorithm*, the fitness can be expressed as the sum of the time it takes the user's fingers to execute certain movements. The paper was written 2002 by formidable researches and was a valuable source because it gave me an idea on how to find a fitness function; however, I, of course did copy his paper. Some limitations were that a lot of incompressible references to Arabic layouts and symbols were made and that the technology used was somewhat outdated. Still, his paper gave me the idea of classifying common finger movements into a set of events.

Using motion events, I created a system to measure the execution time of typing on a keyboard. In this paper, moving any finger from one key to a key horizontally adjacent to that key is referred to as a ‘side-tap’ and has a relative execution time (or execution weight) of the Euclidean distance the finger travels. This distance is proportional to the pixel distance, so the distance between keys in pixels was used for giving each motion event its weight. This was a reasonable idea, since the time it takes to move a finger is largely dependent on the distance it has to move. For the keyboard optimization problem, 8 distinct motion events, each with their own execution time, were used to calculate the fitness of a keyboard.



Event Name	Relative weight (pixel distance)
tap	0
hop-tap	137
leap-tap	284
near-dtap	165
far-dtap	239
leap-dtap	345
side-tap	133
pinky-tap	268

Fig. 4. Visualization of motion events on a general keyboard with a table of pixel distances.

As an example, typing the word H-E-L-L-O on a QWERTY keyboard will induce the following motions (fingers are initially positioned in their base-position before typing the world, e.g. left index is on 'F,' right index is on 'J').

Motion Event	Key movement (on QWERTY)	Event Time
side-tap	J -> H	133
hop-tap	D -> E	137
tap	L -> L	0
tap	L -> L	0
hop-tap	L -> O	137

Fig. 5. Table of motion events induced by typing "HELLO" on a QWERTY keyboard.

Thus, typing HELLO is encoded with the motion pattern:

side-tap, hop-tap, tap, tap, hop-tap

Therefore, the fitness of this individual is

133 + 137 + 0 + 0 + 137 = 407

So in general, the fitness for a keyboard object, *k*, given a text *T* that consists of *n(T)* characters and T_i is the i^{th} character, is

$$f(k,T) = \sum_{i=1}^{n(T)-1} eventTime(k, T_i, T_{i+1})$$

Where *eventTime(k, c1, c2)* is the function that returns the time (or weight) it takes to move a finger from character *c1* to character *c2* on the keyboard layout *k*. Now that a representation of a general solution and a fitness function have been established, the optimization process can begin. The goal is to find an individual *i* that *minimizes* the function, *f(i, T)*, for some text, *T*. It is important to emphasize that we are trying to find a global *minimum* (minimum time to type a text), so a lower fitness value corresponds to a better keyboard layout. For computing the fitness function, over 10mb (10 million characters) of English short stories, books, poetry, and political documents from Project Gutenberg were used.

The Approach

Focusing on optimizing the three main rows of keys -- a total of 33 keys -- yields 33 factorial layout combinations, or 8.7×10^{36} layouts! Thus, testing all solutions (using brute force) is entirely infeasible. Since evaluating the fitness of one keyboard layout takes around 2 seconds, the total time it would take to brute force the best solution would be $2 \cdot 8.7 \times 10^{36} = 1.7 \times 10^{37}$ seconds, or 5.5×10^{29} years -- 4.0×10^{19} times the age of our universe! Instead, a GA is used to effectively traverse this large search space in a negligibly small fraction of time -- approximately two hours. The illustration below outlines the algorithm used. The steps of the algorithm are examined in detail afterward.

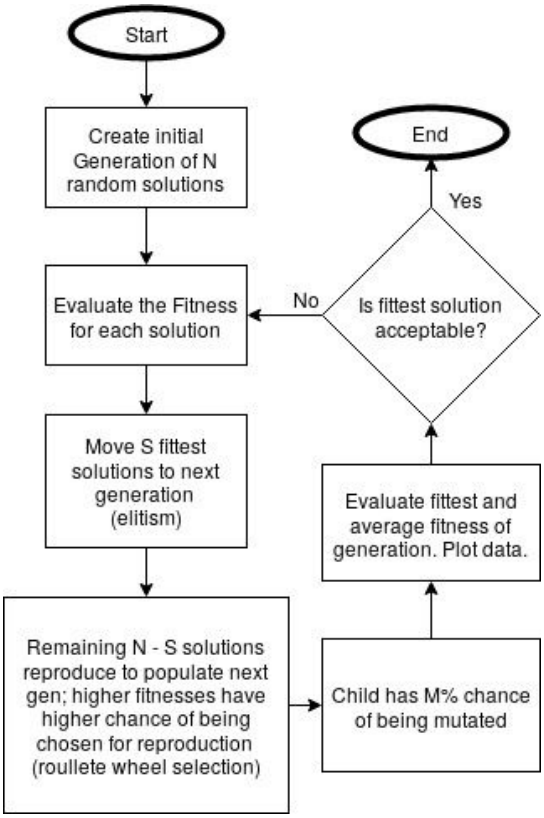


Fig. 6. Flowchart outlining the Genetic Algorithm used. The implementation of the above algorithm and relevant code for each step is located in the appendix.

Initialization

N organisms are created to populate an initial generation, similar to the initial population of bacteria on prehistoric Earth (to relate GAs with Darwin's Evolution). The number of organisms in a generation determines the execution time of the algorithm: too large populations may take too long to evaluate, and too small populations may result in pre-converging, sub-optimal solutions (Levin). Finding the best population size depends on the specific scenario. In the initialization step, *N* randomly generated solutions are created and placed in the initial generation: all subsequent solutions will arise from these initial genes (Levin). After multiple trials, I found that a good population size that balances time and fitness for the keyboard problem ranges from 700 - 1000.

Selection

In the selection phase, solutions compete based on their fitness to reproduce and pass their traits to the next generation. Ensuring that the algorithm continues to move in the right direction, the *S* fittest solutions are copied directly to the next generation. This guarantees that the *S* fittest individuals in the next generation are at least as fit as the fittest individual in the previous generation, e.g. that there is no chance for degradation in fitness (Alabsi). The unconditional copying is called elitist selection, or elitism. The downside of elitism is that if *S* is too large, premature convergence might occur due to a lack of genetic variety. Therefore, *S* should be kept below 30% of the population size according to Alabsi. In the case of the Keyboard Problem, I found that 10% is enough, otherwise the algorithm pre-converges.

To populate the remaining *(N - S)* solutions of the next generation solutions are selected to reproduce and put their offspring into the next generation. There are multiple ways to *select* 'parents', the most prominent one being fitness proportionate selection, also known as Roulette Wheel Selection (Alabsi). In Roulette Wheel Selection (RWS), the probability of an individual being chosen for reproduction is directly proportional to its fitness, e.g., the fitter the solution, the higher the probability for it being selected for reproduction.

Essentially, the probability, *P(i)*, of an individual, *i*, being selected from a population of *N* individuals is the fitness of *i* divided by the total (cumulative) fitness of the population, or

$$P(i) = \frac{f(i)}{\sum_{j=1}^{N} f(j)}$$

where *f(k)* returns the fitness of the keyboard vector, *k*. This method can be visualized as a roulette wheel, where fitter solutions occupy greater portions (have larger slivers) on the wheel. With it, the GA models the biological principle of 'survival of the fittest.'

However, using RWS based purely on the fitness of an individual introduces some drawbacks, as it can quickly kill genetic diversity since it introduces bias for individuals with abnormally high fitness values. For example, consider the case in which there are a few individuals with high fitnesses compared to the rest of the population. These high fitness values completely dominate the roulette wheel; if one individual has a probability of perhaps 70% and the rest of the solutions only get a 1-2% chance, the next population will consist mainly of the genome that had a 70% selection chance. In a matter of a few generations, that single genome will dominate the population, preventing the GA from effectively searching the entire solution space. The event of a few solutions dominating a population is called premature convergence on sub-optimal solutions and occurs if there is too little genetic diversity in the gene pool, resulting in parents not being able to create genetically superior children (Levin).

To make premature convergence less likely, I implemented Rank Selection (RS). The idea of RS is not to consider an individual's absolute numerical fitness, but rather, its fitness relative to other individuals in the population (Arabsi). In RS, the population (of size *N*) is sorted according to fitness and every individual is given a rank. In my program, I let the *most* fittest solution have a rank of 0, the second fittest a rank of 1, and so on, until the *least* fittest has a rank of *N*. From there, RWS is used to come up with the probability of being selected, e.g. an individual's sliver of the roulette wheel now depends on the rank (instead of the fitness). Therefore, in a population of size *N*, the selection probability is the sliver of the individual with rank R, over the sum of all slivers, which can be written as

$$P(R) = \frac{1 - \frac{R}{N}}{\sum_{r=0}^{N-1}(1 - \frac{r}{N})}$$

The below diagram visualizes the differences in probability between Fitness Proportionate Selection and Rank Selection by applying the above formulas to simple cases where N=4 and fitnesses are 80, 15, 3, and 2.

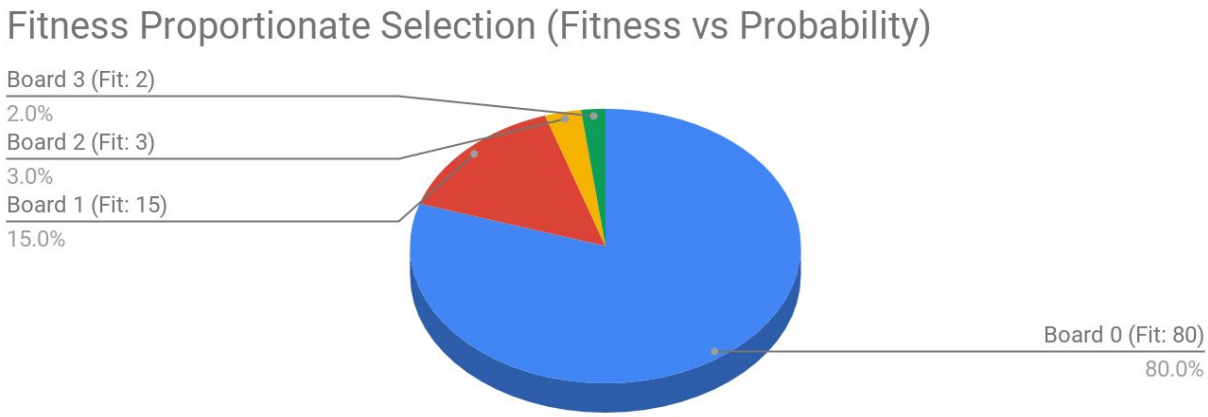


Fig. 7. Roulette Wheel before ranking (probability is directly proportional to fitnesses). Notice how a few individuals dominate the generation.

Fig. 8. Roulette Wheel after ranking (probability is proportional to the rank of an individual). Notice how weaker individuals are given a greater chance to pass their genes, ensuring genetic diversity.

See Appendix 2.1 for Implementation of Selection.

Levin's paper on the "Design and Implementation of Genetic Algorithms for Solving Problems in the Biomedical Sciences" proved a valuable source for understanding different selection methods and GAs. A student at Harvard, Levin writes about GAs and then applies one to a specific biomedical case. He walks the reader through the process of constructing a GA, providing detailed explanations throughout. However, the case he applies it to is rather obscure and complex, and his being a student detracts slightly from the paper's authority. Nonetheless, its formal use of vocabulary and ideas greatly helped me design my GA.

Crossover

Crossover is the next vital step in the GA, as it mimics the concept of nature's reproduction. In crossover, genetic material from one 'parent' is combined, or *crossed over*, with the genetic material of another 'parent' to create offspring. This 'child' produced by the parent is genetically similar to both parents. An example of a "mother" and "father" keyboard crossing over is shown below:

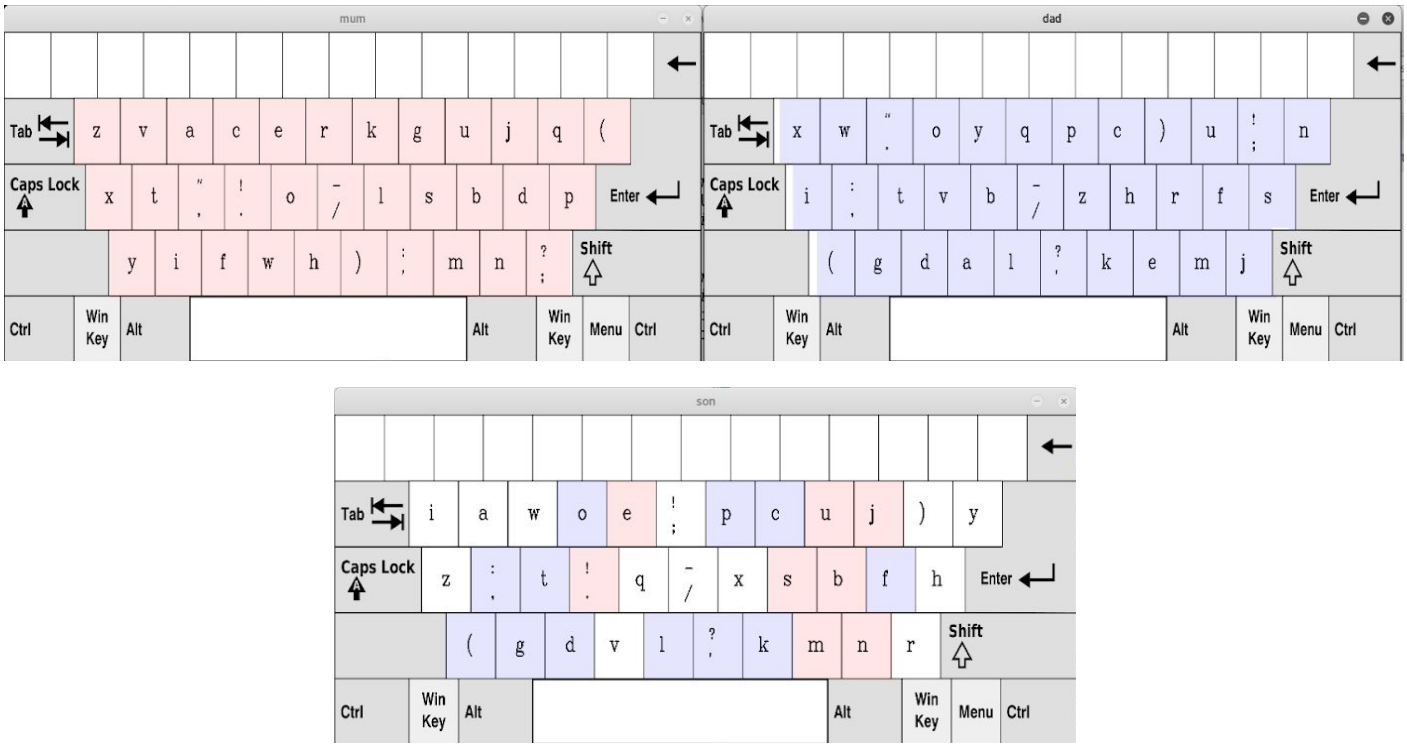


Fig. 9. Crossover simulation. Note the similarities between the offspring and parents.

The above is an example of permutation encoding crossover, where genes are exchanged between parents randomly, yet still ensuring that all genes are available in the created child, so that no keyboard keys are duplicated or deleted (Malhotra). Another method of crossover which can be applied to a larger set of problems is 'value encoding crossover,' where one point is chosen as the crossover point and the child receives one section of DNA from its mother and the other section from its father (Malhotra). However, this does not make a large difference in the greater context of GAs. The implementation of crossover generally does not affect the performance of the algorithm, as long as it mixes the genes of parents in some fashion. This is a recurring theme in GAs -- small changes in a procedure do not greatly change the outcome of the algorithm as long as the procedure still accomplishes what is needed to some extent. Essentially, no pre-defined implementation rules or exactness must be followed for the algorithm to work. Refer to Appendix 2.2 for my implementation of crossover.

Mutation

Mutation is the final genetic operation performed on a generation. Mutation introduces randomness into the gene pool so that the algorithm has a chance to traverse the solution space and avoid local minima (Levin). A common approach to implement mutation -- and the approach used in this paper -- is permutation encoding mutation, where two elements of a vector are swapped at random. For the Keyboard Problem, two keys are chosen randomly and swapped.

Just like in crossover, multiple methods lead to the same result: the introduction of inherent randomness in the generation to increase diversity among solutions. However, one must be wary about the frequency of mutation. The probability of mutation of a gene in a given individual is called the Mutation Rate. If the mutation rate is too big, chances are the offspring will be too random, essentially annulling the work of the GA. According to Malhotra, a mutation rate of 1% - 2% is ideal in most scenarios. To further minimize the risk of creating solutions with too much randomness, not all individuals undergo mutation -- each individual is given a chance of mutation, dictated by the Mutation Probability.

Now that the algorithm is has been explained, here is a summary:

I. Initialize random population of size *N*

II. Allocate a nextGen List

III. Copy the *S* Elite individuals from pastGen into nextGen

IV. For the remaining *(N-S)* individuals,

   A. Rank each individual and conduct Rank Selection to select two Parents

   B. Crossover Parents to create Child

   C. Child has *P%* chance of mutating

      1. If mutating, each key has *M%* chance of being swapped with another key

      2. If not mutating, continue

   D. Place Child into nextGen

   E. Repeat step IV. until the size of nextGen equals *N*

V. Plot fittest individuals in nextGen

VI. Set pastGen equal to nextGen.

VII. Go to step II until it has been repeated *G* times.

Variable	Definition	Example
N	Population Size	800
S	Elitist Survival Size	80
P	Mutation Probability	10%
M	Mutation Rate	2%
G	Number of Generations	60

Implementation in the Appendix.

Results

Running algorithm with a population of size 800, a 10% Elitist Survival Size, a 2% chance for mutation and a 10% chance for each key to be swapped yielded a generation vs. fitness graph that looks this:

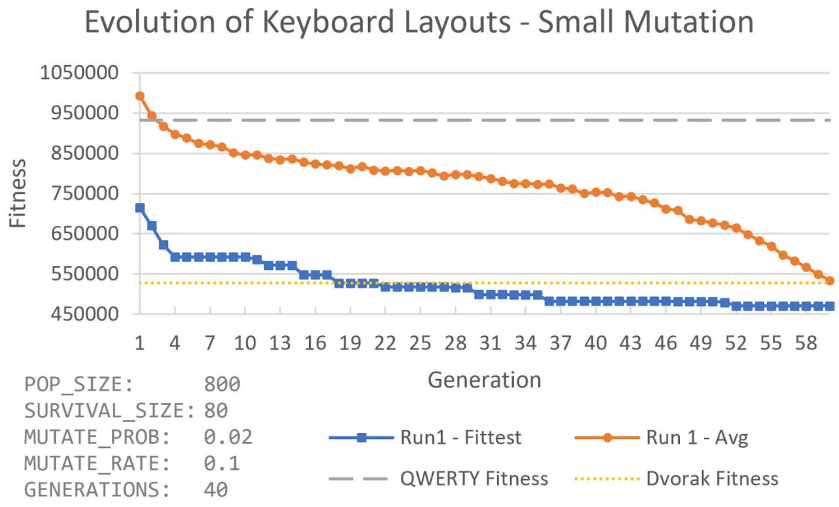


Fig. 10. Trial run with small mutation probability.

and a corresponding keyboard layout with fitness 470,000. Recall that the lower the fitness value, the better the keyboard is. The QWERTY layout has a fitness of 930,000; Dvorak, a fitness of 530,000. So the solution produced is theoretically faster. This is how the keyboard looks like:

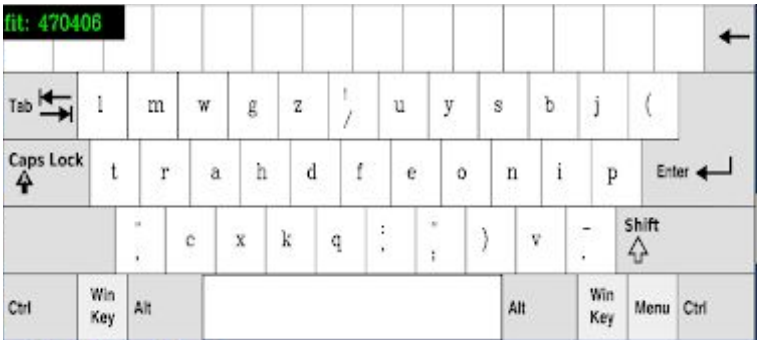


Fig. 11. Run 1 keyboard solution visualized.

Now another run shows how solutions with similar fitnesses are not unique. The below was run with a larger elitist selection rate (30% instead of 10%); the results look different.

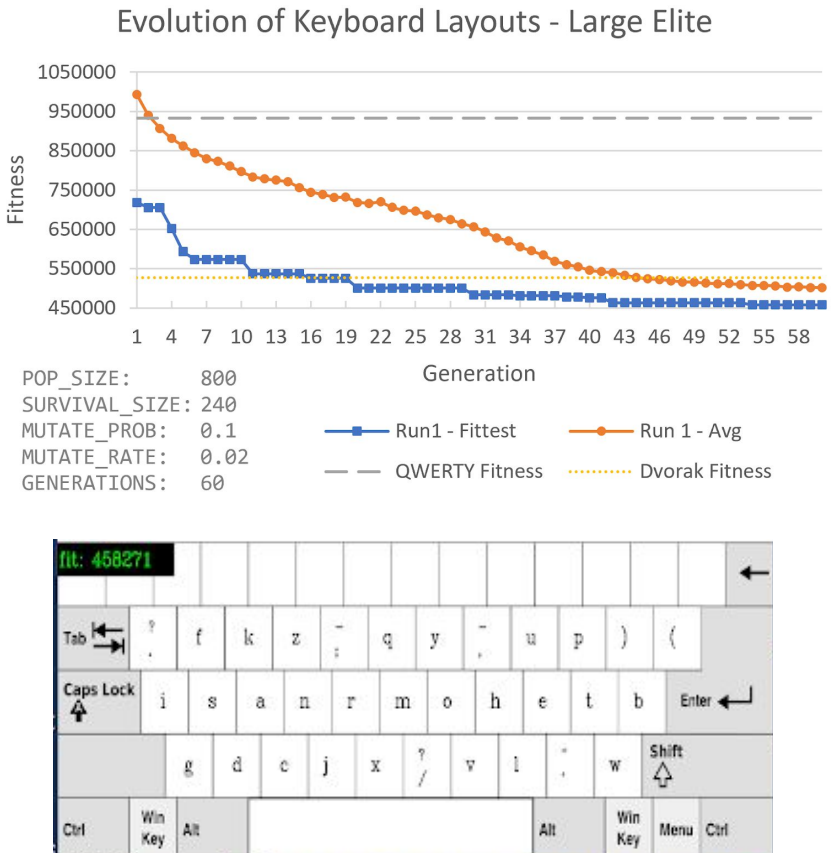


Fig. 12. Graph and corresponding solution of Run 2 with large Elitist Survival Size.

In both trials, the average fitness converges with the minimum fitness over time; however, the speed at which they converge is different because of the different Elitist Survival Sizes and Mutation Probabilities. Using a larger elitist selection produced a slightly better fitnesses, demonstrating that different parameters create different solutions. Comparing these fitnesses to each other, the table on the next page outlines the quality of the best keyboards produced. Other trial runs that created less fit solutions are located in the Appendix.

Layout	Fitness	% Relative Effectiveness
QWERTY	932642	0%
Dvorak	527406	43%
Run 1 (small mutation)	470406	50%
Run 2 (large elite)	458271	51%

Fig. 13. Table with fitnesses relative to QWERTY (better solutions have lower fitness).

The GA ran for about 130 minutes per run on a 3.4Ghz quad-core laptop and yielded solutions that are theoretically over 50% faster than the QWERTY layout and around 8% faster than the Dvorak layout. The data on the graphs show an apparent decrease in the fitness of the keyboards over generations, emphasizing how the solutions at first quickly but then slowly evolve to a near optimum. Interestingly, the average fitness of the population converges with the minimum fitness as the genetic diversity of the population decreases over time. When they fully converge, it means that a close-to-optimal solution has been found and that the algorithm cannot progress any further.

Looking at the above results, however, emphasizes that solutions are not unique, nor guaranteed to be optimal. The produced keyboards look quite different; however, the still have some similar features. Frequently used keys are clumped on the middle row (as one might expect). Depending on the initial conditions of the algorithm though, e.g. Population Size, Elitist Survival Size, Mutation Probability, Mutation Rate, and Number of Generations, the end result will vary. Thus, these trials demonstrate how GAs are useful for finding *good* solutions to large-scale optimization problems, but they are not useful for finding a single best solution. In the appendix, are more trial runs with different parameters, showing the effects of different parameters.

Conclusions

Some may argue that metaheuristics lack theory and mathematical support and point out disadvantages of Evolutionary Algorithms. It is true that from the data it is evident that randomness plays a large role in the ultimate solution to a problem: generating the initial population seems to introduce a large bias in the outcome of the fittest solution. Furthermore, the initial parameters strongly dictate the final solution, so finding which parameters yield the best solution is a problem in itself. Should a large elitist size be matched with a small mutation probability? Or perhaps a larger population size altogether? Finding an answer to these questions would be an interesting area to study further -- could maybe a second GA be developed to optimize the parameters of the first?

In any case, it stands that one cannot know with certainty whether the optimal solution has been found. Solutions can only be compared relative to other solutions (comparing the fittest solution to the fitness of QWERTY for example), making it difficult to evaluate the success of the research method in real life. For instance, the fitness function did not take into account human error and relied on the Euclidean distance between keys to weigh the cost of motion events. These are some limitations to consider. Given the scenario, however, the GA was successful since layouts faster than Dvorak were produced in the simulation universe. Possible ways to improve the fitness function could be to gather data experimentally through human typists to analyze typing behavior and to test different weights for motion events to see how that might change the quality of solutions.

Returning to the research question, GAs can optimize the keyboard layout for speed to a fair extent. The research conducted and the experimental results support that more efficient keyboard layouts can be found using a metaheuristic approach with GAs instead of a more historical, mathematical method. Despite the GA's success, there is a large uncertainty when evaluating the fitness of a keyboard through simulation.

GAs and metaheuristics still prove very helpful in approaching problems with unknown analytical solution methods, or problems that become unfeasibly large quickly like the Keyboard Problem. Essentially, all one needs to account for is a cost/objective/fitness function to optimize any problem. And, if given enough time, a good, perhaps even optimal solution can be found. In

that sense, GAs are useful but not ideal. If a more direct or analytical approach can be used, it should be preferred, as it can in almost all cases produce a solution in less time. For complex problems where analytical solutions are infeasible though, GAs prove to be forerunners in metaheuristic optimization techniques.

Works Cited

Alabsi, Firas, et. al. “Comparison of Selection Methods and Crossover Operations Using Steady State Genetic Based Intrusion Detection System.” *Journal of Emerging Trends in Computing and Information Science,* vol. 3, no. 7, 2012.

Gandomi, Amir H. “Metaheuristic Algorithms in Modeling and Optimization.” *Metaheuristic Applications in Structures and Infrastructures.* Elsevier, 2013. 1-25. Print.

Hempstalk, Kathryn. “The Great Keyboard Debate: QWERTY versus Dvorak.” n.p. 2016.

Korošec, Peter. *Metaheuristic Optimization Algorithms.* n.p. n.d.

Levin, Michael. “Design and Implementation of Genetic Algorithms for Solving Problems in the Biomedical Sciences.” Tech. Medical School, Harvard. n.d.

Malas, Tareq & Taifour, Sinan & Abandah, Gheith. “Toward optimal Arabic keyboard layout using genetic algorithm.” *International Journal of Design Engineering.* 2002.

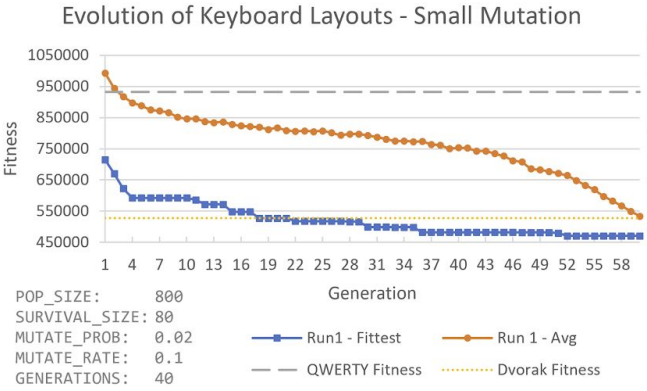
Malhotra, Rahul, et. al. “Genetic Algorithms: Concepts, Design for Optimization of Process Controllers.” *Canadian Center of Science and Education.* 2001.

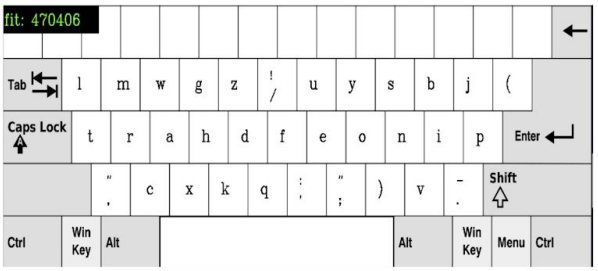
Project Gutenberg. *Project Gutenberg.* 2016.

Appendix

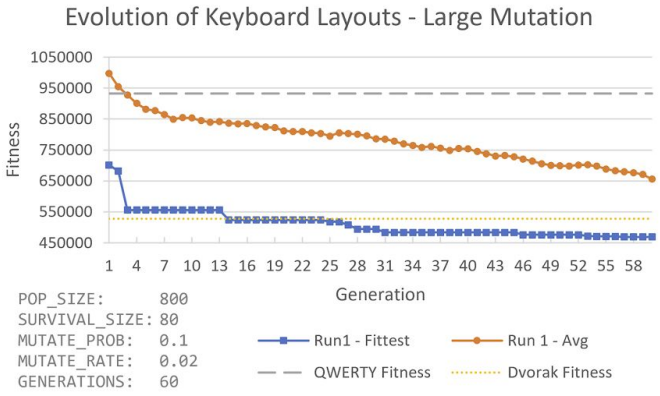
Trial Runs with varying parameters.

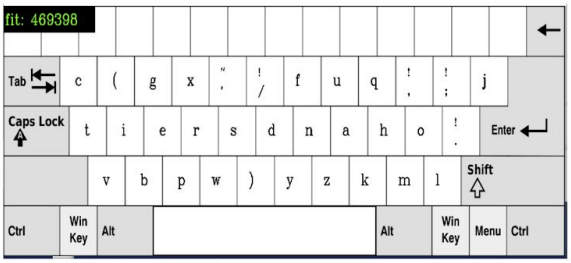
1.0. *Small Mutation Probability Run*



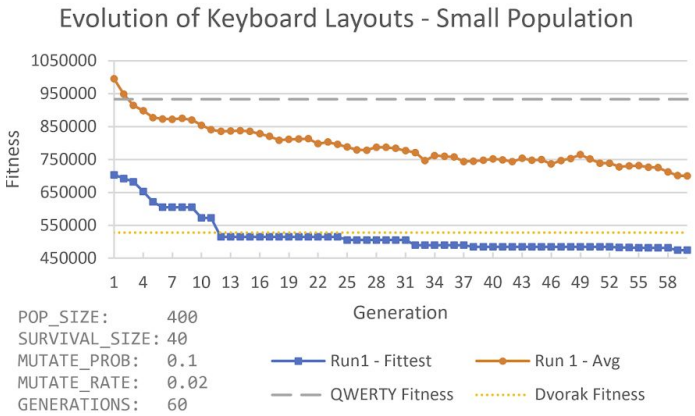


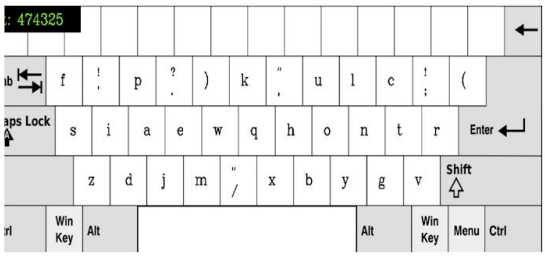
1.1. *Large Mutation Probability Run*



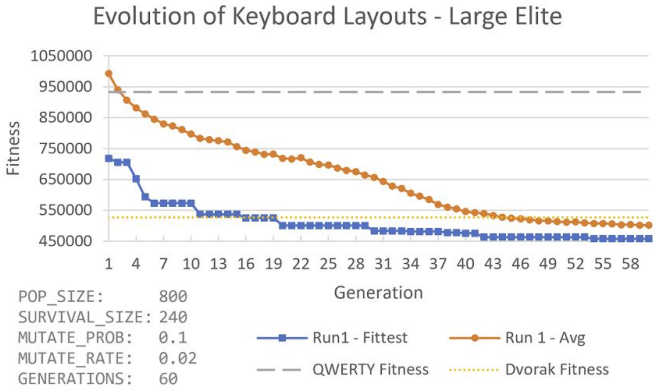


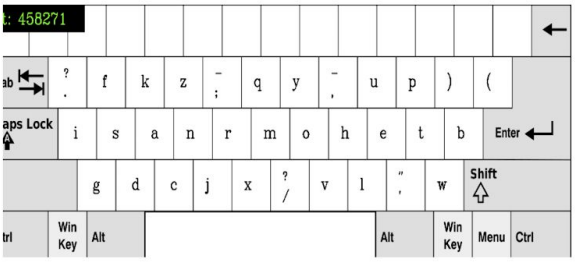
1.2. *Small Population Size Run*



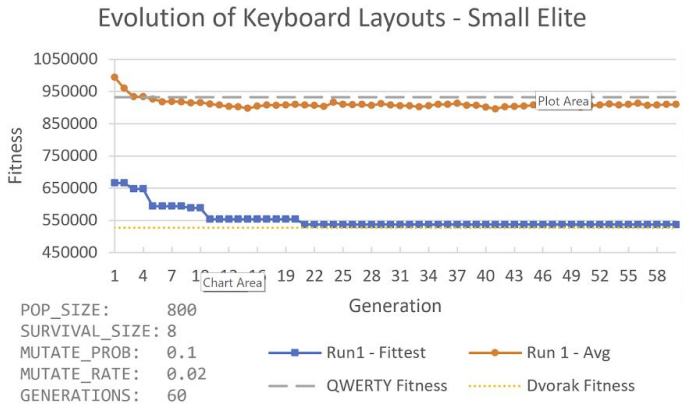


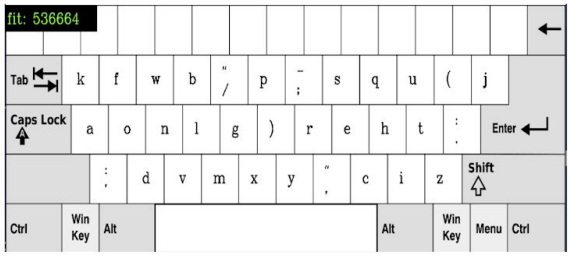
1.3. *Large Elitist Selection Size Run*





1.4. *Small Elitist Selection Size Run*





Selected pieces of code central to the Genetic Algorithm.

2.0. *Initialization*

```
// GA parameters
public class EvolutionParams {
    public int POP_SIZE = 750;     // number of keyboard in a population (P)

    public int SURVIVAL_SIZE = 75;     // default: 30% of P. determines how many
                                       // unaltered species make it to next gen. (S)
    public double CROSSOVER_PROB = 0.5;
    public double MUTATION_PROB = 0.02; // likelihood of invoking the mutation
operator
    public double MUTATION_RATE = 0.1; // likelihood of a gene changing
    public int GENERATIONS = 40;
    public String BOOK_STRING = "res/books/short.txt";
}

// ... //

// INITIALIZATION -- create next generation
LinkedHashMap<Keyboard, Double> nextGen = new LinkedHashMap<>(evoParams.POP_SIZE +
1);
```

2.1. *Implementation of Rank Selection & Elitism*

```
// Below loop: start Rank selection and do Elitism
double inc = 1 / (double)evoParams.POP_SIZE;
double sum = 0.0;

// pastGen is the list of individuals in current population with their fitnesses
Iterator<Map.Entry<Keyboard, Double>> it = pastGen.entrySet().iterator();

// loop through population from fittest to least fittest
for (int i = 0; i < it.hasNext(); i++) {
      Map.Entry<Keyboard, Double> entry = it.next();

      // ELITISM
      // if individual is within the top elitists, copy it directly to next gen
      if (i < evoParams.SURVIVAL_SIZE)
            nextGen.put(
                  entry.getKey(),
                  (double)entry.getKey().getFitness(textManager)
            );
```

```
		// RANK SELECTION -- ASSIGNING SLIVER SIZES
		// rank individual and assign it probability of selection based on its rank
		// biggest sliver size of the roulette wheel corresponds to biggest prob
		// of being selected. Max sliver = 1, min sliver = 1 / POP_SIZE
		double sliverSize = 1 - (i * inc);
		sum += sliverSize;
		entry.setValue(sliverSize);
	}

	// fill remaining N - S population with new solutions (offspring)
	while (nextGen.size() < evoParams.POP_SIZE) {
		Keyboard mother = null, father = null, son = null;

		// ROULETTE WHEEL SELECTION
		// randFather and randMother are random positions on the roulette wheel
		double randFather = Math.random() * sum; // [0, 1)
		double randMother = Math.random() * sum; // [0, 1)

		double cumuSum = 0.0; // cumulative sum. Keeps track of where on the wheel
	we are

		// loop through the wheel until we reach randMother and randFather
		for (Map.Entry<Keyboard, Double> entry : pastGen.entrySet()) {

			if (cumuSum > randMother && mother == null)
				mother = entry.getKey();

			if (cumuSum > randFather && father == null)
				father = entry.getKey();

			if (mother != null && father != null)
				break;

			cumuSum += entry.getValue();	// add fitness to cumusum
		}

		// genetically cross father and mother
		son = reproduce(father, mother);

		// child has random chance to be mutated
		if (Math.random() <= evoParams.MUTATION_PROB)
			mutate(son);

		// put son into next gen
		nextGen.put(son, (double)son.getFitness(textManager));
	}
```

2.2. *Implementation of Crossover*

```
// Implementation of crossover
public Keyboard reproduce(Keyboard dad, Keyboard mom) {
        if (dad == mom)
                System.out.println("Warning: Incest has occurred!");

        // father and mother will exchange genetic material with each other,
        // so don't edit original
        Keyboard father = dad.clone();
        Keyboard mother = mom.clone();

        // copy a random amount of this keyboard to the child,
        // then copy a random amount of the mother's keyboard
        // into this child.
        for (int i = 0; i < Keyboard.NUM_ABCKEYS; i++) {
                if (Math.random() < evoParams.CROSSOVER_PROB) {

                        // get key objects at index i
                        Key fatherKey = father.getKey(i);
                        Key motherKey = mother.getKey(i);

                        // get key indexes of where to switch keys to
                        int fatherKeyIndex =
                              father.asciiToIndex(motherKey.getMainChar());
                        int motherKeyIndex =
                              mother.asciiToIndex(fatherKey.getMainChar());

                        // update first set of keys
                        father.setKey(i, motherKey);
                        mother.setKey(i, fatherKey);

                        // update other set of keys
                        father.setKey(fatherKeyIndex, fatherKey);
                        mother.setKey(motherKeyIndex, motherKey);

                        // re-calculate key indexes for asciiToIndex()
                        father.populateAbcToIndex();
                        mother.populateAbcToIndex();
                }
        }
        // return one child
        return father;
    }
```

2.3. *Implementation of Mutation*

```
public void mutate(Keyboard board) {
	for (int i = 0; i < Keyboard.NUM_ABCKEYS; i++) {
	// if mutation rate met, mutate gene by switching it with a random gene
	if (Math.random() < evoParams.MUTATION_RATE) {
		Key boardKey = board.getKey(i);
		Key otherKey = board.getKey((int)Math.round(Math.random() *
(Keyboard.NUM_ABCKEYS -1)));
		int otherKeyIdx = board.asciiToIndex(otherKey.getMainChar());

		board.setKey(otherKeyIdx, boardKey);
		board.setKey(i, otherKey);

			// re-index keys
			board.populateAbcToIndex();
		 }
	 }
}
```