Spanish Keyboard Layout

David Gabriel Palmerin Morales

Lunes 29, Abril 2019

1. Introducción

Actualmente existen diversos teclados de computadora diseñados para obtener una mejor experiencia para escribir.

El teclado `QWERTY` está diseñado para lenguajes que provienen del Latín, por lo que abarca muchísimos lenguajes, uno de ellos es el Español.
A pesar de que `QWERTY` es el teclado más usado para este tipo de lenguajes, se han desarrollado otros teclados con el fin de mejorar la escritura en un conjunto de lenguajes. Sin embargo, no se encontró algún teclado diseñado exclusivamente para el Español, por lo que se abordará este tema a lo largo de este proyecto.

La técnica de optimización será por medio del *Algoritmo Genético* el cual se describirá su implementación y su funcionamiento posteriormente.

2. Objetivo

Se plantea cómo propósito principal encontrar un teclado para una escritura más óptima para el lenguaje Español. Se definirá una función *fitness* necesaria para el Algoritmo Genético y con base a esta función se comparará el teclado obtenido contra el que todos conocemos, el teclado `QWERTY`

3. Algoritmo Genético

El algoritmo genético es una buena opción de optimización para este problema, incluso aún cuando el espacio de búsqueda es de tamaño `30!` pues consideramos 30 teclas destinadas a uso para letras. Sin embargo, si consideramos únicamente la longitud del alfabeto de español, tendremos `27!`
Se dividirá esta sección en tres secciones: *Genoma*, *Operadores Genéticos* y *Función de Costo*

3.1. Genoma

Sabemos que la representación de nuestro problema en el contexto de algoritmos es genéticos es de gran importancia.

3.1.1. Representación

El genoma será un arreglo de dos dimensiones de tamaño `3` $\times$ `10`, donde `3` es el número de renglones y `10` el número de columnas.
Se escoge esta representación pues es la estructura usual de un teclado, en particular, en el Español únicamente usamos `27` teclas de las `30` disponibles debido al alfabeto que usamos. Dicho esto, el teclado que obtengamos cómo resultado tendrá `3` espacios en blanco distribuidos en cualquier posición que se desee.

Esta representación es suficiente para poder desarrollar el algoritmo genético, sin embargo, también se usa un *diccionario* donde tenemos lo siguiente:

- *Llave* k: k es una letra del abecedario en Español.
- *Valor*: Se almacenará un arreglo `[h, x, y]` donde
  - `h` es `0` o `1`, dónde el `0` denota que la letra k se escribe con tal mano. Esto se puede obtener a partir del genoma pues `h == 0` *syss* k está alguna columna `[0-4]`. Y será `h == 1` *syss* k está en alguna columna `[5-9]`
  - `x` es el renglón donde está la letra k
  - `y` es la columna dónde está la letra k

Gracias al diccionario podemos hacer búsquedas más rápidas que son útiles en la implementación, de esta forma no recorremos el arreglo buscando la tecla k

3.2. Operadores Genéticos

Se desribirán los dos operadores genéticos: *Recombinación* y *Mutación*

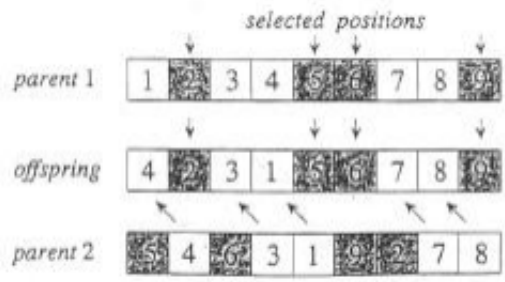
3.2.1. Recombinación

Dado que es indeseable que un teclado repita letras entonces nos encargaremos de realizar la recombinación sin repeticiones, otra posibilidad es penalizar al teclado en la función que definiremos posteriormente, sin embargo se optó por restringir el espacio de búsqueda en este operador.

Dados dos teclados genéticos $\texttt{T}_1$ y $\texttt{T}_2$, nos interesa generar dos hijos que sean obtenidos a partir de los teclados mencionados.

Tomaremos el teclado $\mathtt{T}_1$ y seleccionaremos k teclas de forma uniformemente aleatoria, sin reemplazo. Estas k teclas se colocarán en la misma posición en el teclado hijo $\mathtt{H}_1$, por lo que éste ya tiene k teclas fijas que preservan el orden del padre $\mathtt{T}_1$.
Hecho esto, entonces tomaremos el teclado $\mathtt{T}_2$ buscaremos preservar el orden de éste padre, por lo que recorremos el teclado (genoma) cómo se hace de forma usual el recorrer un arreglo de dos dimensiones, y colocaremos la tecla que corresponde en un espacio en blanco del teclado $\mathtt{H}_1$
De forma análoga para $\mathtt{H}_2$ lo que haremos será el mismo procedimiento pero se fijarán esta vez las posiciones de $\mathtt{T}_2$ y después se llenan los espacios disponibles con $\mathtt{T}_1$

Figura 1: Operador de recombinación[2]



La recombinación se realiza con probabilidad p, por default se define `p = 0.85`. Si se obtiene que no se debe realizar recombinación, entonces simplemente se copian los padres $\mathtt{T}_1$ y $\mathtt{T}_2$

3.2.2. Mutación

La mutación se realiza en cada una de las teclas del genoma, incluyendo aquellas teclas donde no tenga una letra. Esta probabilidad para cada tecla se define por default `p = 1`

Este operador es muchísmo más sencillo, se recorre el genoma (arreglo) y en cada tecla, si cumple la probabilidad `p`, entonces intercambiará posición con otra tecla escogida uniformemente aleatoria, en particular, puede hacer intercambio con ella misma. Hay que recordar que debe actualizar la posición en el arreglo y en el diccionario de tal teclado.

3.3. Función de Costo

La función de costo se basa en el análisis de frecuncia de *monogramas* y *bigramas*.
Estos datos se obtienen a partir de un texto plano en Español y se calculan las frecuencias con el software *Word Creator*, el cual se puede obtener en la siguiente liga:

https://es.sttmedia.com/wordcreator-descarga

Se eligió cómo texto de muestra un fragmento de *El Quijote* a partir del cual se obtienen los monogramas y bigramas usando el software mencionado.
El usuario puede modificar estos archivos los cuales se encuentran en el código fuente:

- `data/text-spanish`
  Contiene el fragmento de El Quijote.
- `data/spanish-monograms`
  Contiene los monogramas.
- `data/spanish-bigrams`
  Contiene los bigramas.

Si el usuario quiere modificarlos deberá respetar el siguiente formato para `spanish-monograms` y `spanish-bigrams`:

`Monograma/Digrama ␣ # de Frecuencias ␣ Porcentaje`

Los monogramas y digramas deben ser representados con mayúsculas.
Además todas las líneas deberán tener `enter (\n)` al final, incluyendo la última línea

3.3.1. Load Distribution

En el artículo en el que se basa este proyecto se muestran que tipo de carga debe tener cada tecla, con carga nos referimos al dedo que debería de presionar una tecla de acuerdo a su posición en el arreglo. Estas distribuciones están basadas en su posicción independientemente de la tecla que contenga, por lo que es útil para el Español. Tenemos la siguiente tabla de cargas:

Tomemos un monograma m el cual tiene coordenadas `[x,y]` en el teclado genético en cuestión, por lo que diremos que :

$$f_m^{opt} = w_f \cdot cargas[x] \cdot cargas[y]$$

Donde w_f denota la carga en cada mano, en la implementación se eligió $w_f = 5$, es decir, ambas manos se usan por igual.
Por otro lado, denotaremos a la frecuencia del monograma m como: f_m
Finalmente, daremos la siguiente función de error, la cual buscamos acercarnos lo más posible al óptimo.
Debido a que es una función de error, nos gustaría *minimizarla*

Figura 2: Tabla de cargas óptimas [1]

No.	*Row (%)*	*Column (%)*	*Finger type*
0	10.87	15.38	Little
1	13.04	10.26	Little
2	15.22	15.38	Little
3	43.48	23.08	Little
4	10.87	17.95	Ring
5	6.52	6.41	Middle
6	-	5.13	Index
7	-	3.85	Index
8	-	2.56	Thumb

$$v_1 = \Sigma_{m \in M}(f_m - f_m^{opt})$$

Implementación `Layout.py`
Funciones: `__ideal_load_distribution__`, `load_distribution`

3.3.2. Key Number

También es importante el número de teclas presionadas en relación a la longitud de la palabra, sin embargo, en tenemos menos letras que teclas, por lo que cada letra puede tener su propia tecla y representar al caracter sin necesidad de usar `shift, alt, etc`. Dicho esto, entonces el *score* es constante.
Un caso especial es para los acentos pues necesitamos oprimir dos teclas, sin embargo, para cualquier teclado que generemos siempre necesitaremos dos teclas para una letra con acentos (excepto Ñ), esto debido a que el diseño del teclado no contempla vocales con acentos cómo otro elemento del alfabeto. Dado que es lo mismo para todos los teclados podemos ignorar una ponderación en este aspecto, por lo que se dejará constante:

$$v_2 = 1$$

3.4. Hand Alternation

Se desea que en un teclado se obtenga la mayor cantidad de alternancia de manos al escribir, esto debido a que permite escribir más rápido debido al desplazamiento que permite en una mano mientras la otra presiona una tecla.

$$v_3 = \Sigma_{d \in D_1} f_d$$

Donde f_d es la frecuencia del digrafo d; D_1 contiene los digramas que se teclean con la misma mano, es decir, supongamos que:

$$d = \alpha_1 \alpha_2$$

Con α_i monogramas. Entonces, usando el genoma de diccionario podemos obtener `[h1,x1,y1]` y `[h2,x2,y2]` que se ha explicado su significado en la sección *Genoma*. Entonces D_1 contiene aquellos digramas donde $h_1 == h_2$. Buscamos que v_3 sea menor pues indica que hay menos digrafos que usan la misma mano, es decir, tenemos un problema de minimización.

Implementación `Layout.py`
Funciones: `__hand_alternation__`

3.5. Consecutive usage of the same finger

Al escribir también sucede que hay palabras que requieren usar la misma mano e incluso usamos el mismo dedo, por lo que esto reduce la velocidad de escritura. Nos gustaría evitar esto por lo que nos gustaría minimizar la siguiente función:

$$v_4 = \Sigma_{d \in D_2}(f_d \cdot dist(d))$$

Donde d es un digrama en el conjunto de Digramas D_2 que contiene los digramas escritos por la misma mano y el mismo dedo.
Por otro lado, *dist* es la distancia manhattan, por lo que entre menor sea la distancia, mejor. Sea d el digrama:

$$d = \alpha_1\alpha_2$$

Con α_i monogramas, nuevamente, con ayuda del genoma obteneremos `[h1, x1, y1]` y `[h2, x2, y2]`, entonces la distancia manhattan será entre los puntos `(x1, y1)` y `(x2, y2)`
Claramente nosotros no desplazamos los dedos cómo lo haría la distancia manhattan, sin embargo, es una distancia pesimista para este contexto y nos ayuda cómo cota superior.

Implementación `Layout.py`
Funciones: `__same_finger_and_hand__`

3.6. Avoid Big Steps

Se busca evitar tecleos hechos por la misma mano y que además genere posiciones raras con los dedos que causen que el flujo de escritura sea menos uniforme y entorpezcan el uso del teclado. Se define la siguiente función:

$$v_5 = \Sigma_{d \in D_3} k(i,j) \cdot f_d$$

Donde D_3 contiene los digramas tecleados por la misma mano, pero no por el mismo dedo y que además la distancia vertical entre dos teclas es mayor o igual a uno.
La función $k(i,j)$ es una función con *penalización* para las peores posiciones de los dedos *i,j*, los cuales representan los dedos óptimos para cada monograma del digrama. Como se mencionó, es una función de penalización, por lo que el objetivo es minimizarla.

Figura 3: Tabla de coeficiente de penalización [1]

Finger	*Thumb*	*Index*	*Mid*	*Ring*	*Little*
Thumb	0	0	0	0	0
Index	0	0	5	8	6
Mid	0	5	0	9	7
Ring	0	8	9	0	10
Little	0	6	7	10	0

Implementación `Layout.py`
Funciones: `__hand_alternation__`

3.7. Overall Score

Dado que todos los *scores* anteriores son problemas de minimización, entonces tendremos en genral un problema de minimización para la combinación lineal de estos atributos.

$$V = \Sigma_{j=1}^{5}(\frac{\gamma_j}{v_{j,ref}} \cdot v_j)$$

- v_j es cada *score* de los mencionados anteriormente.
- γ_j y $v_{j,ref}$ son ponderaciones obtenidas del paper en el que se esta basado el proyecto.

Figura 4: Tabla de ponderaciones [1]

Index	*Relative weight* (γ_j)	*Score of* v_{jref} *(standard)*
Load distribution (v_1)	0.45	0.0341
Key number (v_2)	0.5	1.0000
Hand alternation (v_3)	1	0.4980
Consecutive usage of the same finger (v_4)	0.8	0.1401
Avoid big steps (v_5)	0.7	1.4516
Hit direction (v_6)	0.6	0.1632
Overall score (v_{ref})	-	4.0500

4. Mejoras

Se proponen varias optimizaciones y rediseño:

1. Ocupar las tres teclas restantes para vocales con acento, de esta forma podríamos posiblemente mejorar el costo si asignamos las teclas a vocales con acento que tienen mayor frecuencia. Tendremos que considerar que el *score* de *Key Number* ahora no será constante. Se propone la función de costo a minimizar:

$$v_2 = \Sigma_{d \in V} f_d \cdot f_{d_i}$$

Donde V es un conjunto de digramas. Cada digrama tendrá al menos un monograma con *tilde* y este monograma no deberá tener tecla especial en el teclado en cuestión, es decir, para escribir tal monograma se necesitarán al menos dos tecleos.

Entonces, f_d representa el porcentaje de uso del digrama d y f_{d_i} representa el porcentaje de uso del monograma que tiene la tilde. Claramente buscamos minimizar la penalización dando preferencia a aquellos monogramas con tilde de mayor frecuencia para que tengan su propia tecla.

2. Estos teclados se generan por medio de análisis estadísticos de frecuencias, sin embargo, me gustaría implementar una simulación de escritor *Typer*, de tal forma que pueda medir que tan bueno es un teclado a partir de un conjunto de palabras. Se encargaría de evaluar *alternancia de manos, lejanía de teclas, uso de mismo dedo, etc.* De esta forma podemos obtener otra forma de evaluar el teclado genético obtenido `vs. QWERTY`

3. Que sea posible extender este programa a evaluar conjuntos de lenguajes provenientes de la misma familia. Considero que tomar lenguajes de la misma familia podría facilitar que los teclados obtenidos para el conjunto de lenguajes sigan siendo más útiles que `QWERTY`. Lo primero que intentaría sería obtener un teclado genético que sea bueno tanto para Español cómo para Portugués, sin embargo, esto tiene varias dificultades, la primera es la diferencia de alfabetos y acentos pues el Portugués tiene mayor cantidad de acentos. Sería interesante ver qué tanto cambia un teclado óptimo simultáneamente para dos (o más) lenguajes que un teclado dedicado exclusivamente a un lenguaje.

4. Probar nuevos procesos de selección donde se aproveche mejor el fitness de cada solución.

5. Buscar mejores métodos de recombinación que preserven la unicidad de teclas y que ayuden a converger más rápido a una solución óptima.

5. Implementación

Se agrega el código fuente del algoritmo genético el cual contiene los siguientes archivos:

1. `GeneticLayout.py`
   Se encarga de la labor del algoritmo genético cómo la selección de padres, recombinación y mutación de los padres, selección de nueva población y la evolución misma del algoritmo.

2. `Layout.py`
   Modelación del teclado, encontramos la representación del genoma por medio de diccionario y arreglo (`Layout.coordinates y Layout.design`, respectivamente), así cómo de las operaciones de recombinación y mutación. Además, cuenta con las funciones de evaluación mencionadas para obtener la función *fitness* principal.

3. `Utils.py`
   Contiene diversas funciones que son útiles para `Layout.py` y `GeneticLayout.py`

4. `data/*`
   Cómo se mencionó anteriormente, contiene la información estadística de *monogramas* y *digramas*, así cómo el texto plano de donde estos se obtienen. Estos se *n-gramas* se pueden modificar cómo se menciona en la sección *3.3 Función de Costo*

5.1. Ejecución

El programa se debe ejecutar con el intérprete `Python3`

Existen dos tipos de ejecución:

- *Con parámetros*: Tendremos el siguiente formato

  ```
  python3 GeneticLayout.py numero_generaciones numero_genomas
      recombinación_probabilidad mutación_probabilidad
  ```

  Si se escoge esta opción se deberán colocar todos los parámetros, será inválido colocar solo algunos de ellos.

- *Sin parámetros*: Podemos ejecutar de la forma siguiente

  ```
  python3 GeneticLayout.py
  ```

  El cual se traduce a parámetros por default

  ```
  python3 GeneticLayout.py 100 100 0.85 0.10
  ```

Referencias

[1] Emad Khorshid, et al. *A new optimal Arabic keyboard layout using genetic algorithm.* International Journal of Design Engineering, Enero 2010.

[2] *Recombination Operators.* http://mat.uab.cat/~alseda/MasterOpt/GeneticOperations.pdf

Figura 5: Ejecución 1

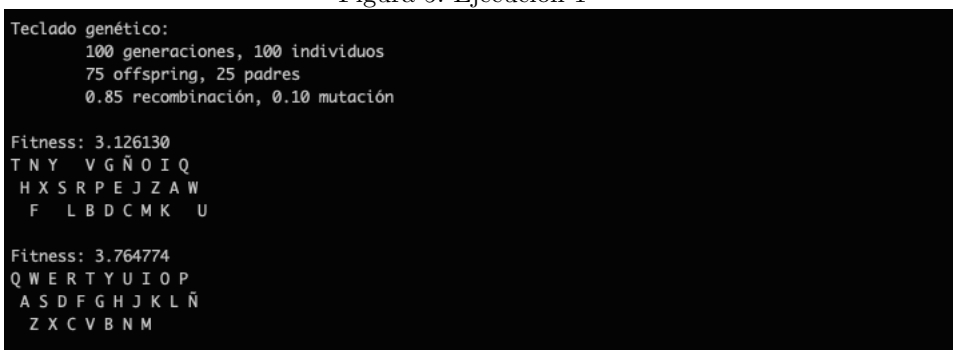


Figura 6: Ejecución 2. Más individuos y generaciones

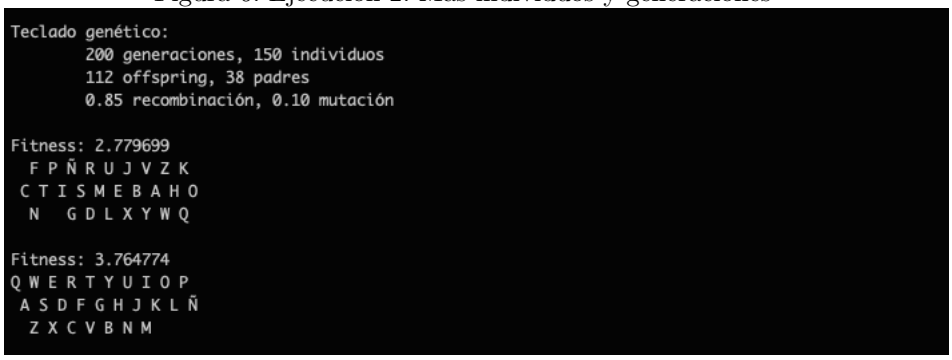


Figura 7: Ejecución 3. Aún más individuos y generaciones

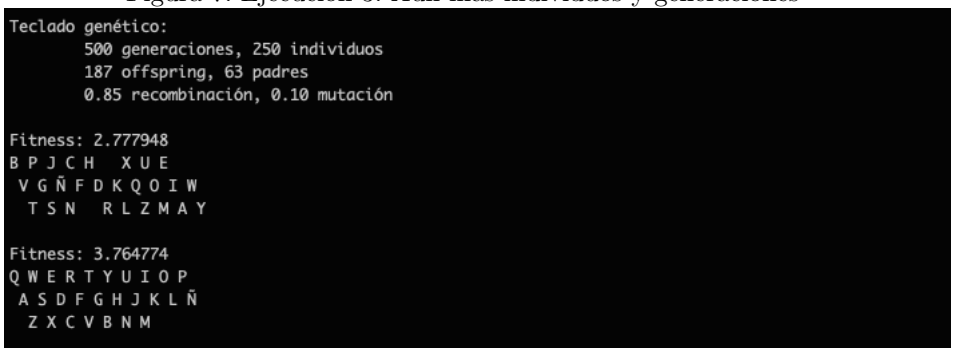


Figura 8: Ejecución 3. Mucho más individuos y generaciones

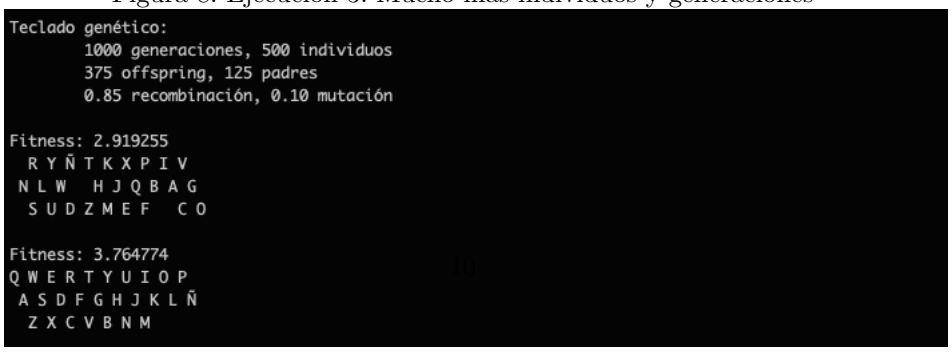


Figura 9: Mejor solución de 20 ejecuciones con parámetros default