Nombre: Francisco López Franco / 156349
Ricardo Edward Meadowcroft / 174006
Renata Monsalve Rubí / 176371
Profesor: Kuri Morales, Ángel.
Materia: Fundamentos matemáticos de la computación

Generador de máquinas de Turing

Introducción

El siguiente trabajo es un generador de máquinas de turing. Recibe un documento de texto que, tras convertirse a una cadena binaria, representa el output que se desea genere la máquina de Turing tras procesarse. El código hace uso del primer proyecto de la materia, el cual recibe una máquina de Turing (en su representación binaria) y la acciona para procesar una cinta, de tamaño variable, que inicia en ceros. Tras generar una máquina y comparar su output contra el de la cinta objetivo, se hace uso de algoritmos genéticos para mejorarla y conseguir máquinas cuya cinta de output es tiene cada vez una mayor tasa de coincidencia con la cinta objetivo.

Para entender los resultados obtenidos en el proyecto es necesario que primero evaluemos el marco teórico, la forma en la que se implementa el algoritmo y el flujo del código y, finalmente, cómo interactuar con él a través de la interfaz implementada.

Manual Teórico

Generar máquinas de Turing puede ser resuelto a través de diferentes algoritmos. Sin embargo, en parte dada su eficiencia y en parte por curiosidad como ingenieros en computación, se optó como equipo por implementar un algoritmo genético. Entendidos de forma general son búsquedas heurísticas que obtienen su nombre ya que imita la teoría de la evolución natural: de una población, se reproducen los más aptos para obtener una nueva generación que tiene como punto de partida al mejor “poblador” y se cambian los elementos más débiles de la población actual por los mejores elementos de la nueva generación. De este modo se hace una mejora paulatina e incremental de los resultados y de la población en general.

Pero para poder implementar código que imite la selección natural es necesario comprender de forma previa y a profundidad las etapas que componen el proceso. La teoría que se va a revisar no es la única forma de implementar algoritmos genéticos, estos son un campo de estudio y cuentan con diferentes variantes, sin embargo, aquí se describe la información que compete a nuestra implementación.

La selección natural puede ser dividida en 5 etapas. La primera corresponde a la población inicial. Esta población tiene un genotipo, compuesto a su vez por genes, y un fenotipo. En las máquinas de Turing su genotipo es el código binario que las compone. Tienen siempre una longitud múltiplo de 16 debido a que cada estado que compone la máquina se representa en 8 bits (en el extremo izquierdo) que se ejecutan si el input en la cabeza de la máquina es cero y 8 bits (en el extremo derecho) en caso de que el input sea 1.

De ese conjunto de 8 bits, los primeros dos son el output que se imprime en la cinta y si el movimiento es a la derecha (0) o a la izquierda (1). Es importante mencionar esto ya que ello implica que se pueden tener las 4 combinaciones de dos bits y todas son parte de una máquina válida. Los 6 bits restantes son el siguiente estado de la máquina; dado que con 6 bits se pueden conseguir hasta 64 combinaciones binarias, este será el número máximo de estados que tenga la máquina que se genere (nótese que una máquina de Turing puede tener N estados, por lo que la limitante es por un parámetro implementado en el código y no de las máquinas en sí mismas).

Cada poblador tendrá por tanto un genoma de hasta 16 (bits) * 64 (estados); la longitud de su cinta es de hasta 1024 bits y cada uno de estos bits es considerado un gen. Dado que cualquier combinación de 8 bits es un output, movimiento y siguiente estado válido, se pueden mutar todos los bits y aún así obtener la representación de un estado válido. Con ello se concluye que no es necesario sesgar la posición de los bits mutados o intercambiados ya que la nueva máquina seguirá siendo una máquina válida.

Por otro lado, el fenotipo de las máquinas puede ser considerado como sus "características físicas", es decir, la tabla de estados que la componen, el output que genera y su valor de fitness. El término nos lleva a la segunda etapa de la selección natural: la función fitness o función de aptitud. Para poder reproducir al más apto debemos antes definir qué se entiende por apto. Esto será determinado por una función que arroje un puntaje al evaluar a cada poblador y luego definir qué se considera como el mejor puntaje. Por ejemplo, puede darse un puntaje inicial y penarlo para hacer reducciones para después escoger al más apto buscando los puntajes más altos de la población, emulando un sistema de maximización y escogiendo al menos penado de los elementos.

Podría también ser una función que calcule la diferencia entre la cinta del output y la deseada y el puntaje de cada elemento sea el número de bits erróneos. En ese caso se buscaría al más apto haciendo una búsqueda de mínimos. Se decidió intentar al menos dos funciones de fitness en la implementación del código, estas serán revisadas en el manual técnico.

Tanto la búsqueda es de mínimos o máximos (o cualquier otro método), esta corresponde ya a la etapa de selección. Se decide el número de "mejores elementos" que serán seleccionados y que funcionarán como los "padres" de la siguiente generación. Cabe resaltar que en el proceso se quitan los elementos más débiles de la población y se intercambian por los nuevos mejores. Así la población tiene siempre el mismo tamaño pero conserva sólo a sus mejores elementos. En el caso de que los mejores de la

nueva generación obtengan puntajes de fitness más bajos que los peores de la población actual, no se realiza el intercambio, ya que ello generaría un detrimento de los genomas y no una mejora.

La cuarta y quinta etapa son intercambio o crossover y mutación. Estas son las dos formas en las que se producen los cambios de los genotipos (una vez que ya se seleccionaron los padres). Cruce funciona marcando puntos a lo largo del genoma (imágen 1.1) de los dos padres e intercambia los bits de los dos padres (imagen 1.2). El resultado son dos nuevos elementos que comparte una parte de su genoma con cada padre (imagen 1.3). Se pueden realizar cruces en varios puntos para un mismo hijo o se pueden realizar cruces en varios puntos y cada uno genera un par diferente de hijos.

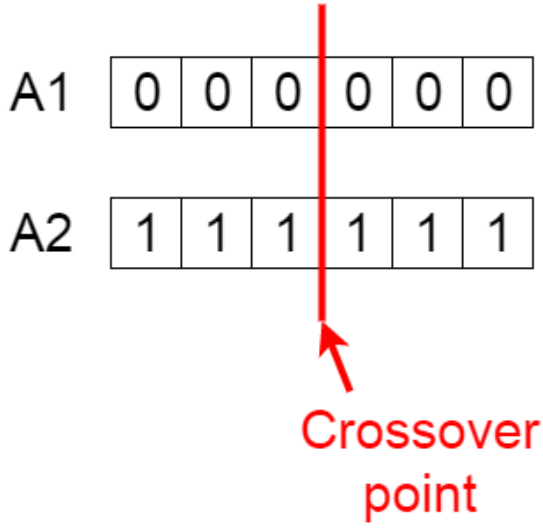


Imagen 1.1 Punto de cruce

Sin embargo, el cruce sólo permite el intercambio de bits preexistentes, la población tenderá a hacerse homogénea con el paso de las generaciones. Por ello se requiere la inclusión de las mutaciones (que forma parte del proceso evolutivo natural que imita), que implemente cambios exógenos aleatorios y disminu-

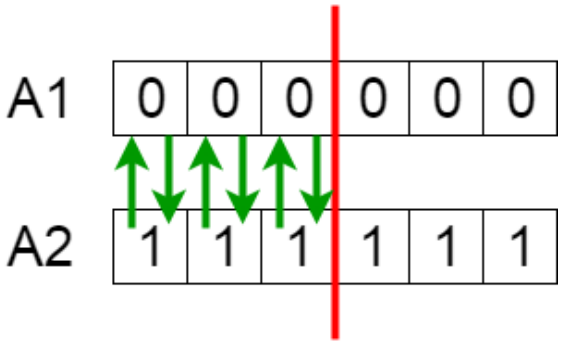


Imagen 1.2 Intercambio de bits

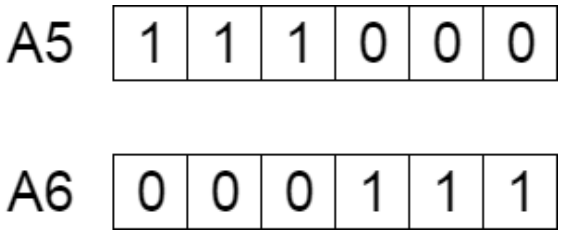


Imagen 1.3 Generación resultante

ya esta convergencia de los individuos. Dado que es una cadena binaria, este proceso simplemente implica la inversión de un bit (0 a 1 y viceversa) de un conjunto de bits en una posición aleatoria.

Las primeras generaciones inducen cambios más grandes, considerando que los padres iniciales suelen tener un genotipo muy diferente entre ambos. Sin embargo, el cruce (crossover) continuo genera homogeneidad, como ya se mencionó, y la población tenderá a converger: sus genomas irán perdiendo diversidad y se parecerán cada vez más entre ellos, esto sucede a pesar de que se le integran mutaciones. Entre mayor sea la similitud de los padres es mayor la probabilidad de que el cruce intercambie dos conjuntos de bits idénticos y, por tanto, los hijos resultantes sean iguales a ambos padres sin que se haya generado un cambio en el genoma.

Todos los factores mencionados explican por qué el fitness que se consigue

en las primeras generaciones difícilmente encontrarán una mejora notoria en las siguientes; un buen resultado dependerá en gran medida de que los padres de la primera generación ya tengan un valor de fitness cercano al óptimo. Se puede, sin embargo, aumentar la probabilidad de mutación y la cantidad de bits que son invertidos, esto aumenta el número de generaciones que requiere para converger, o puede no converger del todo, pero también aumenta la diferencia entre los genotipos de los padres y los hijos, es decir, se aleja del genotipo que hasta el momento es el más apto.

Manual Técnico

El código incluye las siguientes clases:
Main: Esta clase es la encargada de instanciar las clases correspondientes y llamar los métodos necesarios para la manipulación de las poblaciones y empujar las siguientes N generaciones que serán creadas. Tras ello, reporta los resultados para que el usuario pueda evaluar la máquina conseguida. No contiene métodos propios y corre en un loop infinito que permite al usuario decidir hasta qué punto desea trabajar con la población conseguida o salir (y reiniciar el proceso) para cambiar a toda la población de máquinas.

UTM: Retoma de la clase codificada para el proyecto previo. Recibe la máquina de Turing y regresa la cinta que produce. Se redujo el código en ella para retirar todas las llamadas a imprimir información en la línea de comandos. El algoritmo de procesamiento se conservó, pero ya no calcula la productividad de la cinta, ni avisa que se ha llegado a un estado de Halt. Dado que este estado es el número 63 (considerando al estado 0 como el primero), puede que la mejor máquina de la población actual se detenga dado que se le acabó la cinta y no porque llegue al estado H. Sabiendo que ambos casos de parada son posibles, se conservó la detención del Halt y se agregó la detención sin Halt.

Printer: Contiene un único método estático que recibe el código binario que representa una máquina de Turing y lo traduce a una tabla para facilitar la lectura por parte del usuario. No se agregó a ninguna de las clases existentes para mantener el orden y propósito asignado a cada clase.

Tape: Incluye la funcionalidad de comparación, es decir, una cinta se puede comparar con otra haciendo un conteo del número de bits que comparten. En un principio, el programa comparaba la cinta objetivo con la obtenida de la máquina para utilizar este conteo de coincidencias en la función de fitness que implementa. Sin embargo, dado que no se estaban consiguiendo valores superiores al 75% como tasa de coincidencia en las pruebas iniciales, tras modificar las probabilidades de crossover y mutación.

Se optó por modificar la función de fitness modificando el valor que se obtiene de los matches. Dado que las cintas se crean con un largo base y se escribe

por sobre de ellas, las comparaciones podrían arrojar coincidencias que no competían a la sección de la cinta que de facto representa la cadena de caracteres deseada. Por ello se mejoró el algoritmo restando del número de coincidencias los unos que se obtienen en secciones que exceden las posiciones donde se ubican los datos reales de la cinta tras pasarse de caracteres a binario.

TuringMachine: Es la segunda clase con mayor funcionalidad e importancia. Incluye el método que genera los primeros pobladores asignando al azar los bits del genoma de cada uno. Para que la función de random no genere en cada población inicial los mismos genomas se tuvo que hacer aleatoria la semilla con la que se instancia el random. Cuenta también con una función que elimina bloques de ceros. Esta se implementa con el propósito de crear máquinas con un menor número de estados, siendo esta característica una del fenotipo que puede servir para granular la función de fitness.

Las demás funciones son básicas para poder presentar la información al usuario, como regresar la máquina como una variable de tipo String, hacer un conteo de los estados usados, eliminar los ceros excedentes al largo real del genotipo de la máquina (recordemos que todas inician con 1024 bits que corresponden a 64 estados), dados los parámetros, asignar un bit a una posición en la máquina binaria y los gets y sets necesarios.

TuringMachineGenerator: Esta clase es la que recibe todos los parámetros solicitados al usuario en la interfaz. Es la encargada de implementar todas las etapas descritas en el marco teórico, por ello, es la única clase cuyo código explicaremos a detalle (todas las previas se encuentran debidamente comentadas y su función no es parte del algoritmo genético).

El primer método que se llama activa una interfaz que tiene predefinidos valores para todas las variables que se involucran en el proceso del algoritmo genético. Se muestran al usuario y se permite que sean aprobados en conjunto o modificarlos. Este es un método estático ya que lo realiza el propio generador previo a ser instanciado. Una vez validados los valores regresa una instancia de sí mismo para poder ejecutar el proceso. Se optó por este flujo con el objetivo de obtener una clase monolítica que no dependa menos del Main para ser creada. Es el único método estático de la clase.

El constructor de la máquina recibe la cinta objetivo, la probabilidad de mutación y crossover, la semilla que usará en sus funciones aleatorias, el tamaño de la población y el número de generaciones que avanza en cada iteración. Asigna los valores recibidos a sus variables internas y crea un arreglo vacío donde almacenará la población y un arreglo paralelo donde guardará el valor de fitness de cada

uno. Hasta este punto no se cuenta con ningún genoma.
Es desde el Main que se empuja a que el generador de máquina de turing cree las siguientes N generaciones, por ello se crea vacío, para evitar duplicar la N.
Al llamar el método que avanza las generaciones este revisa si es la primera generación o una posterior. En caso de ser la primera, usa el método de Tape para crear N genomas aleatorios y computa el fitness de cada una. En el segundo caso llama primero a un proceso de selección que busca las dos mejores máquinas (siguiendo una búsqueda de máximos) para ser usadas como los "padres" de la siguiente generación. Las etapas de la selección que modifican los genomas (crossover y mutación) son llamados únicamente si el random genera un número por debajo de los factores de probabilidad sesgada asignados a cada método. Así, se realizar un "azar controlado" para designar qué combinación de cruce y mutación se realizará. De los hijos resultantes se busca si superan a los dos peores fitness actuales de la población y, en caso de ser así, se reemplazan con los nuevos hijos obtenidos.
Cabe resaltar que el método de selección encuentra a los padres pero no trabaja sobre de ellos, sino que realiza una copia del código y opera sobre esta copia. Se contempla así la posibilidad de que los hijos no superen la aptitud de los padres y no se pierda con ello a los dos mejores pobladores actuales.

La función crossover inicia siempre en cero hace uso del random para marcar el punto de corte del genoma de la máquina (desde cero hasta la longitud total del genoma). Los bits intercambiados tienen por tanto la misma longitud que el número obtenido al azar. Y lo intercambia entre los dos padres, siguiendo la teoría que planteamos en el manual previo.

Mutation también hace uso del random entre cero y la longitud del genoma para invertir un bit del genoma en una posición obtenida. Se probó también con la estrategia de cambiar más de un bit: usando el mismo valor de probabilidad, se recorría todo el genoma y se decidía de forma individual para cada bit si este sería modificado. Dado que el parámetro usado es 0.05f, existía una probabilidad muy baja de que el resultado final del genoma impidiera que la población converja. Tras correr varias pruebas modificando el valor, no se obtuvo un incremento notorio el la tasa de coincidencia, por ello se optó por una versión que genera menor procesamiento y se conservó el primer algoritmo de mutación implementado.

Contiene los get necesarios para operar con su población y las funciones para convertir cadenas de ASCII a binarias y viceversa.

La última de las funciones notorias del código corresponde al cálculo del fitness. Implementa una función que

calcula la aptitud de una máquina de Turing. Corrimos pruebas con el código buscando mínimos y máximos, consideramos (dado los resultados de tasa de coincidencia obtenidos) que es mejor conservar el algoritmo que busca máximos. Para asignar el valor de fitness se consideran tres fenotipos:

La cantidad de coincidencias que tuvo con la cinta meta, valor que se multiplica por 1000 para darle un mayor peso en comparación con los otros factores.

La diferencia de longitudes entre ambas cintas, ya que la máquina puede estar generando cintas cortas que por tanto jamás van a coincidir con la cinta meta. Esta es multiplicada por un peso de 10.
El cálculo de la complejidad de Kolmogorov, a este también se le ha asignado un peso de 10.

En algún punto se le restaban puntos entre más estados tuviera la máquina, sin embargo, esto repercute de forma negativa en los resultados finales. Empujaba a las máquinas a reducirse y ello implicaba que cuando generaban cintas de menor longitud, difícilmente podía volver a incrementar ese valor, estancando la posibilidad de una mejora o un incremento en la tasa de coincidencia.

Manual de Usuario

Al ejecutar la clase Main, se inicia el código. Se piden al usuario los valores para el generador de máquinas en la siguiente secuencia:

1. Se pide la semilla que será usada por el random, se recomienda un valor de de 6 dígitos.

```
> java -classpath .:/run_dir/junit-4.12.jar:target/dependency/* Main

Deme la semilla del generador de números aleatorios:
```

2. Pide el nombre del archivo donde se encuentra la cadena de texto meta. En este caso se recomienda usar dinosaurio.txt y modificar los contenidos del archivo con los caracteres que se deseen.

```
Deme el nombre del archivo de datos que quiere procesar:
dinosaurio.txt
```

3. A continuación, la interfaz muestra al usuario los valores con los que se va a crear el generador y puede aprobarlos todos en conjunto, en cuyo caso escribe 's',

o spedir que se modifiquen, escribiendo '*'.

3b. Si el usuario solicitó modificarlos, la línea de comandos preguntará cada valor conforme el usuario vaya ingresando los datos

```
Numero de individuos:
200
Numero de transiciones:
1200
Long. de la cinta:
2500
Prob. de cruzamiento:
0.95
Prob. de mutacion:
0.05
Numero de generaciones:
2000
```

3c. Una vez ingresados todos, el sistema pide nuevamente que se compruebe si son correctos. Se puede apreciar en la siguiente imagen que se han guardado los cambios en los valores de número de individuos y número de generaciones.

```
1) Numero de individuos:       200
2) Numero de transiciones:     1200
3) Long. de la cinta:          2500
4) Prob. de cruzamiento:       0.950000
5) Prob. de mutacion:          0.050000
6) Numero de generaciones:     2000

¿Modificar (S/*)?
*
```

En caso de reingresar '*' volverá al paso previo. Si, en cambio, se ingresa una 'S', procede a ejecutar las iteraciones de generaciones y entrega los resultados finales. En la imagen se han agregado la totalidad de los estados de la máquina, pero, por motivos de lectura, de aquí en adelante sólo se agregará el primer estado de la máquina.

```
a) Numero de coincidencias: 110
b) Longitud de la cinta de datos: 200
c) Generacion actual: 2000

    ==> Tasa de coincidencias: 0.55

************************************
* La complejidad de Kolmogorov: 688 *
************************************
Hay 42 estados en la Maquina de Turing
 EA | O | M | SE || O | M | SE |
 ------------------------------
  1 | 0 | R | 18 || 0 | R |  3 |
  2 | 0 | R |  7 || 0 | R | 17 |
  3 | 0 | R | 30 || 0 | R | 27 |
  4 | 1 | L | 22 || 1 | L | 26 |
  5 | 1 | L |  8 || 0 | R | 31 |
  6 | 0 | R | 24 || 1 | L | 13 |
  7 | 1 | L |  2 || 1 | L | 28 |
  8 | 0 | R | 31 || 1 | L | 32 |
  9 | 1 | L | 32 || 1 | L | 37 |
 10 | 1 | L | 32 || 0 | R |  3 |
 11 | 1 | L | 14 || 0 | R | 33 |
 12 | 0 | R | 19 || 1 | L | 24 |
 13 | 0 | R | 21 || 1 | L | 26 |
 14 | 1 | L | 27 || 0 | R | 10 |
 15 | 1 | L |  4 || 1 | L | 31 |
 16 | 1 | L | 15 || 1 | L | 36 |
 17 | 1 | L | 41 || 1 | L |  3 |
 18 | 1 | L |  8 || 0 | R | 31 |
 19 | 1 | L | 25 || 0 | R | 18 |
 20 | 1 | L | 22 || 1 | L | 20 |
 21 | 0 | R |  H || 0 | R | 39 |
 22 | 1 | L | 17 || 0 | R | 24 |
 23 | 0 | R | 39 || 0 | R | 31 |
 24 | 0 | R |  9 || 1 | L |  6 |
 25 | 1 | L | 36 || 1 | L | 31 |
 26 | 0 | R | 41 || 1 | L | 36 |
 27 | 0 | R | 40 || 0 | R | 29 |
 28 | 0 | R | 35 || 1 | L | 31 |
 29 | 0 | R |  8 || 1 | L | 14 |
 30 | 1 | L | 20 || 1 | L | 27 |
 31 | 0 | R |  1 || 1 | L | 40 |
 32 | 0 | R | 22 || 1 | L | 19 |
 33 | 1 | L | 34 || 1 | L |  5 |
 34 | 1 | L |  5 || 0 | R | 30 |
 35 | 1 | L |  4 || 0 | R | 26 |
 36 | 1 | L |  7 || 1 | L | 28 |
 37 | 0 | R | 14 || 0 | R |  8 |
 38 | 1 | L | 28 || 1 | L | 36 |
 39 | 1 | L | 40 || 0 | R | 12 |
 40 | 1 | L |  7 || 0 | R |  1 |
 41 | 1 | L |  8 || 0 | R | 37 |
 42 | 1 | L | 39 || 1 | L |  4 |
```

Al concluir pregunta si se desea continuar la búsqueda. Escribir '*' terminará el pro-

ceso. Si se contesta 's', volverá a correr el número de generaciones guardado con los mismos parámetros de probabilidad y volverá a imprimir los resultados.

Resultados

Una vez implementado el algoritmo descrito en el manual técnico se hicieron diferentes pruebas con las variables que recibe la clase generadora de máquinas de Turing. Para ello se corrieron varias pruebas con los mismos valores y cadena usando un método diferente al de la interfaz, se eliminaron las preguntas para agilizar el proceso de correr varias pruebas consecutivas; este se ha conservado comentado.

Las primeras pruebas permitieron establecer un parámetro de comparación en la tasa de coincidencias esperado en 0.7 que representa un 70%. Para asignar los valores iniciales se usaron aquellos que habíamos notado se desempeñaban mejor durante las pruebas realizadas en el proceso de escribir el código y la implementación de nuevas funciones o la modificación de las previas. Sin embargo, una vez concluido el código, se iniciaron las pruebas formales del código; para ello se modificó individualmente cada parámetro para evaluar su impacto directo en el resultado final.

1. Tamaño de población: El valor original con el que se probó fueron 1000 "pobladores". Se modificó el tamaño de la población para que fuese el doble pero tras correr 5 pruebas el valor no se mejoró de forma notoria. Sin embargo, al reducirla a 500 elementos la mejor de las pruebas arrojó un 77% de coincidencia.

```
//parametros del generador, se han de
int seed_length = 100000;
int population_size = 500;
int max_transitions = 1200;
int tape_length = 2500;
float crossover_probability = 0.95f;
float mutation_probability = 0.05f;
int generationsXiteration = 1000;
float ponderation_factor = 0.4f;
```

```
a) Numero de coincidencias: 154
b) Longitud de la cinta de datos: 200
c) Generacion actual: 1000

    ==> Tasa de coincidencias: 0.77

************************************
* La complejidad de Kolmogorov: 672 *
************************************
Hay 42 estados en la Maquina de Turing
```

2. Transiciones máximas: Se inició con el valor numérico "1200". Tras incrementar y reducir el valor y correr 5 pruebas, no se consiguió ningún cambio notorio que denotara que esta variable tiene un impacto sobre los resultados. La mejor tasa de coincidencia alcanzada fue de 73.5% con 500 transiciones.

```
//parametros del generador, se han de
int seed_length = 100000;
int population_size = 100;
int max_transitions = 500;
int tape_length = 2500;
float crossover_probability = 0.95f;
float mutation_probability = 0.05f;
int generationsXiteration = 1000;
float ponderation_factor = 0.4f;
```

```
a) Numero de coincidencias: 147
b) Longitud de la cinta de datos: 200
c) Generacion actual: 1000

    ==> Tasa de coincidencias: 0.735

************************************
* La complejidad de Kolmogorov: 656 *
************************************
Hay 41 estados en la Maquina de Turing
 EA | O | M | SE || O | M | SE |
```

3. Generaciones por iteración. El parámetro estaba inicialmente fijado en 1000, al duplicarlo simplemente se alcanzaban los mismos resultados que con 1000 pero en una o dos iteraciones menos. Sin embargo, el algoritmo arroja un valor inicial en cada prueba que mejora en pequeñas cantidades conforme se avanzan las generaciones debido a la convergencia. Por ello se conservó en 1000.

```
//parametros del generador, se han de
int seed_length = 100000;
int population_size = 100;
int max_transitions = 1200;
int tape_length = 2500;
float crossover_probability = 0.95f;
float mutation_probability = 0.05f;
int generationsXiteration = 1000;
float ponderation_factor = 0.4f;
```

```
a) Numero de coincidencias: 153
b) Longitud de la cinta de datos: 200
c) Generacion actual: 4000

    ==> Tasa de coincidencias: 0.765

************************************
* La complejidad de Kolmogorov: 704 *
************************************
Hay 43 estados en la Maquina de Turing
```

4. Crossover probability. Este valor se inicia en un número cercano a 1 (0.95), sin embargo, a diferencia de los valores previos, la modificación de este generaba un resultado notoriamente diferente, pero inferior al alcanzado con los valores. Por ello, en el caso de este parámetro se corrieron el mismo número de pruebas pero con una mayor variación del parámetro. Se consideró que los valores que ofrecen mejores resultados son aquellos cercanos a 1, por este motivo se conservó el 0.95f.

```
//parametros del generador, se han de
int seed_length = 100000;
int population_size = 100;
int max_transitions = 1200;
int tape_length = 2500;
float crossover_probability = 0.5f;
float mutation_probability = 0.05f;
int generationsXiteration = 1000;
float ponderation_factor = 0.4f;
```

```
a) Numero de coincidencias: 136
b) Longitud de la cinta de datos: 200
c) Generacion actual: 3000

    ==> Tasa de coincidencias: 0.68

************************************
* La complejidad de Kolmogorov: 656 *
************************************
Hay 41 estados en la Maquina de Turing
```

A pesar de que en las pruebas se obtuvo una tasa de coincidencia mayor con 0.2f que con 0.5f, el promedio de los valores obtenidos con 0.2f es de 0.63. Se consideró que el 0.705 conseguido en esta muestra era un caso poco común

```
int seed_length = 100000;
int population_size = 100;
int max_transitions = 1200;
int tape_length = 2500;
float crossover_probability = 0.2f;
float mutation_probability = 0.05f;
int generationsXiteration = 1000;
float ponderation_factor = 0.4f;
```

```
a) Numero de coincidencias: 141
b) Longitud de la cinta de datos: 200
c) Generacion actual: 1000

    ==> Tasa de coincidencias: 0.705

************************************
* La complejidad de Kolmogorov: 656 *
************************************
Hay 40 estados en la Maquina de Turing
```

y podia deberse a la semillam, la cual es la única variable no regulada. Ello planteó un nuevo cuestionamiento sobre las decisiones tomadas en cuanto alm algoritmo implementado.

5. Mutation probability. En el caso de esta variable se notó que tiene un impacto directo en el resultado final, al igual que el de crossover probability; sin embargo, en el caso de este, genera mejores resultados entre más cercano es a 0 y no a 1. Esto quiere decir que permitir a los genomas cruzar secciones de su código trabaja a favor de las siguientes generaciones, mientras que intercambiar bits al azar dentro del genoma genera un detrimento en las generaciones posteriores.

```
//parametros del generador, se han de
int seed_length = 100000;
int population_size = 100;
int max_transitions = 1200;
int tape_length = 2500;
float crossover_probability = 0.95f;
float mutation_probability = 0.9f;
int generationsXiteration = 1000;
float ponderation_factor = 0.4f;
```

```
a) Numero de coincidencias: 109
b) Longitud de la cinta de datos: 200
c) Generacion actual: 1000

    ==> Tasa de coincidencias: 0.545

************************************
* La complejidad de Kolmogorov: 672 *
************************************
Hay 41 estados en la Maquina de Turing
 EA | O | M | SE || O | M | SE |
```

```
//parametros del generador, se han de
int seed_length = 100000;
int population_size = 100;
int max_transitions = 1200;
int tape_length = 2500;
float crossover_probability = 0.95f;
float mutation_probability = 0.5f;
int generationsXiteration = 1000;
float ponderation_factor = 0.4f;
```

```
a) Numero de coincidencias: 109
b) Longitud de la cinta de datos: 200
c) Generacion actual: 1000

    ==> Tasa de coincidencias: 0.545

************************************
* La complejidad de Kolmogorov: 688 *
************************************
Hay 42 estados en la Maquina de Turing
 EA | O | M | SE || O | M | SE |
```

```
//parametros del generador, se han de
int seed_length = 100000;
int population_size = 100;
int max_transitions = 1200;
int tape_length = 2500;
float crossover_probability = 0.95f;
float mutation_probability = 0.2f;
int generationsXiteration = 1000;
float ponderation_factor = 0.4f;
```

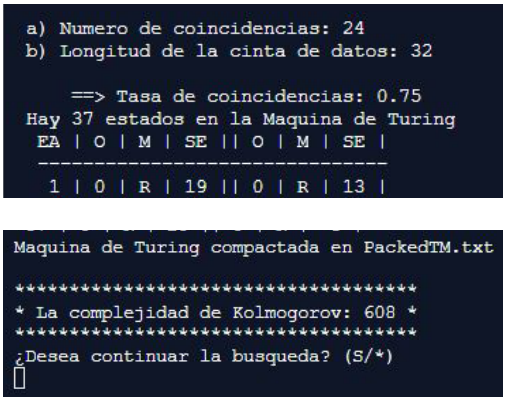
```
a) Numero de coincidencias: 144
b) Longitud de la cinta de datos: 200
c) Generacion actual: 1000

    ==> Tasa de coincidencias: 0.72

************************************
* La complejidad de Kolmogorov: 624 *
************************************
Hay 38 estados en la Maquina de Turing
 EA | O | M | SE || O | M | SE |
```

Una vez que se comprobó que los valores iniciales eran los óptimos, o que se implementaron ajustes de acuerdo a los resultados de las pruebas aisladas, se procedió a probar con ellos los resultados de variar la cadena meta a replicar.

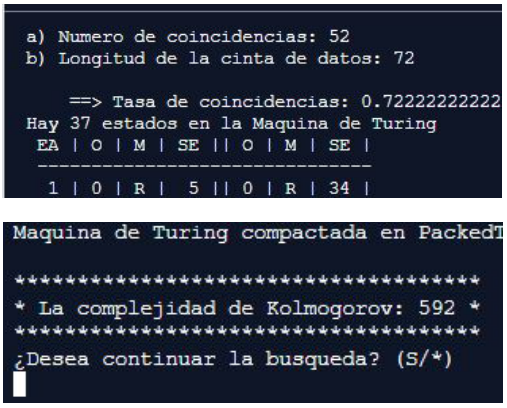
Se decidió que se intentarían 5 diferentes cadenas que varían en longitud y composición. Es decir, se evitarían incrementar las cadenas agregando un mismo caracter reiteradamente o que la cinta se extendiera agregando siempre la misma secuencia de caracteres. Las cintas no incluyen caracteres del alfabeto en español, es decir, se evitó usar vocales acentuadas.

Para cada una de las cadenas se corrieron 5 pruebas y se llevó cada una de esas hasta al menos 10 mil generaciones. Se guardó una captura de pantalla del mejor resultado obtenido de cada cadena y se reporta a continuación. El código imprime la máquina de Turing completa, sin embargo, para evitar ocupar grandes extensiones del reporte con imágenes de código que no suma nada a los resultados finales, se han suprimido los estados intermedios y se muestran únicamente el primer y último estado de cada máquina.

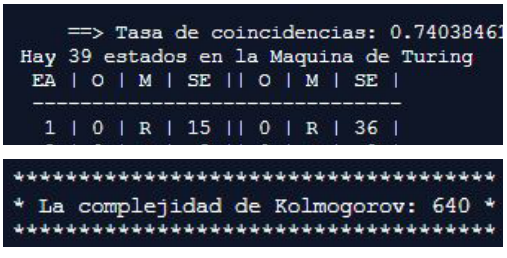
1. Se inició con la cadena “hola” de cuatro caracteres.



2. La segunda prueba se realizó con la cadena “ABCABCABC” de longitud 9 caracteres.



3. La tercera cadena es “el dinosaurio” de longitud 13.



4. Para el cuarto caso se incrementó la cadena previa para hacerla más parecida al cuento original de Augusto Monterroso; “el dinosaurio estaba alli” tiene longitud de 25 caracteres.

5. Para la última prueba se siguió la lógica del inciso previo y se probó la cadena “el dinosaurio todavía estaba allí” de longitud 33 caracteres.

```
a) Numero de coincidencias: 140
b) Longitud de la cinta de datos: 200
c) Generacion actual: 5000

    ==> Tasa de coincidencias: 0.7
Hay 37 estados en la Maquina de Turing
 EA | O | M | SE || O | M | SE |
 -------------------------------
  1 | 1 | L |  6 || 0 | R | 14 |
```

```
************************************
* La complejidad de Kolmogorov: 592 *
************************************
```

Durante las pruebas, a pesar de que la tasa de coincidencia no cambiaba, el valor de la complejidad de Kolmogorov disminuye debido a que el algoritmo puede encontrar máquinas de turing que obtienen el mismo resultado que sus ancestros pero en un número menor de estados.

Por ello, anexamos la captura de pantalla (de dos iteraciones realizadas en la misma prueba; en ellas se puede leer la mejora en el valor de complejidad y la reducción en los estados que la máquina imprime como propios.

```
a) Numero de coincidencias: 121
b) Longitud de la cinta de datos: 160
c) Generacion actual: 5000

    ==> Tasa de coincidencias: 0.7562
Hay 41 estados en la Maquina de Turin
 EA | O | M | SE || O | M | SE |
 -------------------------------
  1 | 1 | L | 38 || 1 | L |  9 |
```

```
************************************
* La complejidad de Kolmogorov: 656 *
************************************
```

```
b) Longitud de la cinta de datos: 160
c) Generacion actual: 7000

    ==> Tasa de coincidencias: 0.75625
Hay 40 estados en la Maquina de Turing
 EA | O | M | SE || O | M | SE |
 -------------------------------
  1 | 1 | L | 37 || 1 | L | 30 |
```

```
************************************
* La complejidad de Kolmogorov: 640 *
```

Se puede ver que la tasa de coincidencia se conserva, sin embargo la generación actual es 2000 unidades mayor que la muestra previa. El número de estados pasó de 41 a 40 y, en consecuencia, el cálculo de la complejidad ha disminuido las 16 unidades que representan los 16 bits del estado que se ha eliminado.

Conclusiones

Debido a que son varios los parámetros que influyen en el resultado final sería interesante generar un mayor número de variantes de código para evaluar los algoritmos de las funciones. Es decir, así como durante el proyecto se probaron algoritmos que buscaban el mínimo y se descartaron, podrían probarse otras formas de contar las coincidencias, o de modificar los algoritmos que manipulan las cintas para volverlo un proceso heurístico. Podrían contarse la productividad de la máquina, hacer corrimientos que eliminen los estados que no son accedidos antes de volver a cruzar, etc.

Incluso en los algoritmos implementados para el crossover y la mutación podríamos implementar otros. modificar

más de una sección del genoma, o un número fijo de genes donde lo aleatorio sea la posición que se modifica. Para la mutación podrían implementarse algoritmos que intercambian bits consecutivos de tamaño y posición variable, o probar mutando en las mismas posiciones para la mejor máquina y la segunda mejor máquina, etc.

Otra limitante de la solución que podría mejorarse a futuro consiste en la semilla que se usa para en random. Imprimir la semilla usada en cada prueba podría haber arrojado información sobre las semillas que convienen y buscar patrones entre la semilla, los demás valores y los mejores resultados. Pero, a pesar de que se evaluó el impacto de las demás variables en el resultado final, la semilla permaneció como un random de Math y no se revisó a fondo qué valores se estaban ingresando, podría depender en menor medida del azar y se podría asignar mayor control a esta variable.

La función de fitness también se presta a diversas implementaciones. No sólo se pueden modificar los valores de ponderación o peso de cada variable, pueden agregarse un mayor número de variables, el número de estados no usados, si contiene o no un estado de Halt o la distancia entre los conjuntos de bits acertados, por ejemplo. Una última mejora que podría agregarse como variable es el número de máquinas que se usan como padres. En esta implementación se usaron 2, sin embargo, podría modificarse para que fuesen 3 o 4, que sigue siendo un número bajo para poblaciones de 200 elementos, pero ofrecería más posibilidades para realizar crossover ya sea entre parejas o entre varios.

Todas estas mejoras son comprensibles debido a la complejidad del tema de los algoritmos genéticos. Sin embargo, tomando en consideración que esta es la primera implementación que realiza el equipo, las pruebas reiteradas permitían eventualmente alcanzar valores superiores a 70%, con un máximo histórico registrado de 77%. En alguna ocasión obtuvimos un valor superior al 80% pero en ese momento aún no se estaban guardando las métricas de las pruebas pues nos encontrábamos en la etapa de desarrollo. Si bien es cierto que un 70% no podría considerarse óptimo, es un excelente punto de partida y un resultado bueno o suficiente dadas las deficiencias y mejoras potenciales ya mencionadas.

De forma general, los algoritmos genéticos y las máquinas de Turing resultan dos temas de la computación muy interesantes: no sólo porque superan las barreras de depender de un único lenguaje de programación, sino porque existe la posibilidad de escalar horizontalmente los proyectos e incluir más modelos matemáticos, factores y variables (como el control de la semilla o un factor de ponderación) y, por ello, se prestan a ser terreno fértil de investigación.