DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo practico final

Metaheuristicas

Segundo Cuatrimestre de 2017

Integrante	LU	Correo electrónico
German Pinzon	475/13	pinzon.german.94@gmail.com
Angel More	931/12	angel21fer@gmail.com

DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Introducción

1.1. Enunciado

En el presente trabajo estaremos resolviendo el problema de Linear Ordering (LOP) utilizando un algoritmo genético. Dada una matriz C de nxn, LOP plantea encontrar una permutación p de filas y columnas tal que el valor de :

$$f(p) = \sum_{i} \sum_{i<j} c_{p(i)p(j)}$$

sea máximo. Es decir, que la suma de la parte triangular superior sea máxima.

1.2. Plan de desarrollo

Hemos decidido utilizar el lenguaje Ruby para desarrollar el algoritmo, debido a que gracias a su simplicidad nos permite enfocarnos en las partes importantes del problema y abstraernos de los detalles de implementación.

Al utilizar un algoritmo genetico es necesario desarrollar las distintas etapas del mismo. Plantearemos en primer lugar, como representar a los miembros de la población (denominados genotipos). Luego, de qué manera vamos a estar generando una población inicial, es decir el conjunto de soluciones factibles al problema del cual se parte. Una vez que tenemos este conjunto, tenemos que definir como generar nuevas soluciones a partir de él, en el caso de los algoritmos genéticos se trata de seleccionar dos individuos (padres) de nuestra poblacion. Para esto experimentaremos con dos criterios de selección distintos: Roulette Wheel Selection y Tournament Selection. Una vez elegidos se realiza un cruzamiento (crossover) entre los padres para generar dos soluciones hijas. Nos basaremos en el metodo Davis' Order Crossover y una variante del mismo.

En las soluciones hijas puede producirse una mutación, una alteración de la solución. En nuestro caso la mutación esta dada como el intercambio aleatorio entre dos filas y las dos columnas correspondientes. Variaremos la probabilidad con la que puede ocurrir este fenómeno para determinar si existe un valor que nos permita obtener mejores soluciones. Al generar nuevas soluciones es necesario eliminar individuos de la población, por lo que se plantea un algoritmo de selección de sobrevivientes.

El objetivo de plantear distintos criterios en cada etapa del algoritmo genético, es para determinar que combinaciones son las mejores a la hora de querer resolver este tipo de problema usando un algoritmo genético. Estos criterios serán aplicados a distintas matrices de prueba. Exhibiremos los resultados obtenidos y por último procederemos a aplicar las mejores combinaciones a nuevas matrices para observar la calidad de la solución.

2. Desarrollo

A continuación desarrollaremos los críterios y algoritmos utlizados en cada etapa del algoritmo genético:

2.1. Representación (genotypes)

Para representar los miembros de la población vamos a estar implementando una clase Genotype (de la cual hablaremos más adelante) que tendrá guardada una lista de índices de filas/columnas que representará las permutaciones, por ejemplo: sea M una matriz a permutar de n x n, y M' la matriz permutada según la lista de permutacion L con:
$L = [e_1, e_2, .., e_{n-1}, e_n]$
Entonces la fila e_1 será la primer fila de la matriz M', la e_2 la segunda y de manera análoga para el resto de los elementos. También tendremos la suma de la matriz triangular superior ($fitness\ value$). Y Además provee funcionalidad adicional como reconstruir la matriz y buscar en base a ciertos criterios dentro de la población, un genotype en particular.

2.2. Población inicial

Sabemos que existen básicamente dos enfoques para generar la población inicial. El primero es mediante la utilización de una heurística y el segundo es generar las soluciones de manera aleatoria. En nuestro caso, decidimos utilizar el enfoque aleatorio debido a que lo ideal es tener una población inicial lo más diversa y amplia posible y si utilizamos una heurística existe la posibilidad de generar soluciones "similares"que perjudiquen a la población inicial en su totalidad.

2.3. Selección de padres

Para la selección de padres el algoritmo permite dos configuraciones posibles: tournament y roulette. El primero toma de la población k individuos al azar y extrae las dos mejores como los padres (aquellos con mayor $fitness\ value$).
El segundo es un poco más complejo, asigna una probabilidad a cada solución de la población, en base a su $fitness\ value$. En nuestro caso a mayor $fitness\ value$ mayor probabilidad. Y luego elige dos elementos utilizando la función de probabilidad correspondiente. La implementación de esta selección tiene algunos detalles como por ejemplo que hay que normalizar los $fintess\ values$ para todos sean positivos, con el fin de calcular correctamente cada probabilidad.

2.4. Crossover

Para el crossover decidimos implementar dos métodos distintos, siendo el primero una ligera variación del segundo. El primer método entra en la categoría de los denominados "multi point crossover". Básicamente se eligen dos números al azar, el primero n_1 entre 0 y la mitad de la dimensión la matriz y el segundo n_2 entre $n_1 + 1$ y la dimensión de la matriz. Si nombramos los padres como p_1 y p_2, la lista de permutacion del primer hijo va a tener en las posiciones del rango $[n_1, n_2]$ a los elementos de p_1 comprendidos entre el mismo rango. El resto de los elementos de esta lista serán los correspondientes a la lista de permutaciones de p_2, respetando ese mismo orden y teniendo especial cuidado en omitir valores repetidos. Luego se repite el procedimiento entero para generar el segundo hijo, pero invirtiendo los roles de p_1 y p_2. En las experimentaciones denominados a esta implementación *crossover normal*
El segundo método, *crossover variado*, no conserva la posición de los elementos que hereda cada hijo de p_1 sino que estaran como prefijo en su lista de permutación.

2.5. Mutación

Para la mutación lo que hacemos es intercambiar dos elementos en la lista de permutaciones. Las mutaciones sucederan con cierta probabilidad, en nuestro caso elegimos 25 %, 50 % y 75 %. Con la hipotesis de que cuanto mayor es la frecuencia de las mutaciones nuestro algoritmo más se parece a una elección totalmente aleatoria.

2.6. Selección de sobrevivientes

Para generar la nueva población a partir de la anterior no utilizamos ningún tipo de aleatoriedad si no que eliminamos a dos de los integrantes en base al valor de la fitnesss function. Decidimos que aquellos dos individuos que arrojen el menor valor sean los eliminados. De esta manera mantenemos constante el tamaño de la población a lo largo de las iteraciones.

2.7. Fitness function

Esta función es la que utilizaremos para determinar que individuos de la población sobreviven. La elección es trivial ya que lo que queremos maximizar es la suma de los elementos de la parte triangular superior y justamente esta operación va a representar la función que queremos.

2.8. Criterio de parada

En cuanto a la condición de corte del algoritmo, decidimos mantener las cosas simples y que sea al alcanzarse un máximo de iteraciones.

3. Implementación

En esta sección vamos a contar un poco como es el diseño del código, cuales son las facilidades para extenderlo en trabajos futuros, como se relacionan las distintas partes y de que manera debemos establecer las distintas configuraciones a la hora de experimentar.

El diseño del código fue realizado de cierta manera para que sea posible extenderlo fácilmente agregando o modificando las distintas configuraciones posibles. A continuación describimos los componentes principales del mismo con el nombre de las clases:

3.1. Clase LinearOrderingSolutionsGenerator

En esta clase concentramos mayormente la lógica necesaria para el esquema básico de lo que es un algoritmo genético. La clase es la encargada de ir modificando la población inicial agregandole elementos, parar cuando se llegue a una cantidad máxima de iteraciones, obtener los padres y los hijos a partir de ciertos criterios externos e ir almacenando cual es la mejor solución encontrada hasta el momento. Cuando decimos çriterios externos"lo que en realidad estamos queriendo remarcar es que esta clase está totalmente desacoplada de lo que son los métodos de crossover, el criterio para eliminar un elemento de la población, la mutación y la selección de padres. Todo esto es configurable externamente y la única responsabilidad que tiene esta clase es generar la población inicial y utilizar estos criterios que se le brindan de manera externa para, como dijimos antes, generar el esquema de lo que es un algoritmo genético.
A continuación el pseudocódigo de como es el método encargado de generar la solución.

```
generar poblacion inicial
for i in 0..max_iteraciones
    padres = criterio_de_seleccion_de_padres.select
    descendencia = criterio_de_cruzamiento.cross
    criterio_de_mutacion(descendencia).mutate
    creterios_de_supervivencia(poblacion)
    agregar_descendencia_a_la_poblacion
end
seleccionar la mejor solución de la población
```

3.2. Clase Genotype

Esta clase es muy importante ya que contiene la representación de los elementos de nuestra población y de ella dependen todos los criterios que estuvimos nombrando anteriormente. Además, provee funcionalidad para obtener la matriz que está representando el genotipo con la lógica necesaria para realizar las permutaciones correspondientes. Provee también los métodos que nos permiten obtener el *fitness value* de un genotipo y también seleccionar en base a un criterio (puede ser el mejor o peor en base al *fitness value* o puede ser en base a otro criterio configurable externamente) un genotipo de la población.

3.3. Criterios

En esta categoría entran las clases FitnessBasedSelection, RouletteParentSelection, TournamentParentSelection, SwapMutation, NormalMultiPointCrossover y VariationMultiPointCrossover. Cualquiera de estos criterios puede ser modificado independientemente del resto del código, también es posible agregar nuevos criterios tanto de selección de padres como de mutación y selección de sobrevivientes. El único requisito es que respeten los métodos que usa LinearOrderingSolutionsGenerator, por ejemplo cualquier criterio de mutación debe tener el método *mutate*, cualquier criterio de selección de padres el método *select*, etc.

3.4. Requerimientos y modo de uso (archivo main.rb)

El algoritmo esta programado en el lenguaje de programcion Ruby por lo que es necesario tener instalado el mismo en su versión 2.3.1 o superior. Se cuenta con un archivo main que requiere los siguientes

parámetros:

- Nombre del archivo que contiene la matriz a permutar, indicando en la primera linea la dimension de la misma y luego la matriz correspondiente.
- tamaño de la población.
- nombre del método de selección de padres, pudiendo ser "*tournament*" o "*roulette*".
- número de elección de padres pósibles, para ser utilizado en la selección de padres.
- nombre del tipo de crossover ya sea : "*normalcrossover*" o "*variationcrossover*".
- la probabilidad de mutación, un número entre 0 y 100.
- el número máximo de iteraciones.

por lo que la linea de comando para la ejecución es:
ruby main.rb p_1 p_2 p_3 p_4 p_5 p_6 p_7
donde los p_i son los parámetros mencionados anteriormente. Finalmente obtendremos un nuevo archivo nombrado como la matriz a permutar con el sufijo $'\_out'$, aquí se encontrará el fitness value para la mejor permutación encontrada, la lista de permutaciones y la matriz permutada.

4. Experimentación

Se ha realizado una extensa experimentación del algoritmo utilizando distintas configuraciones con respecto a la selección de padres, cantidad de iteraciones, tamaño de la población, probabilidad de mutación y tipo de crossover. Lo que hicimos fue tomar pequeños conjuntos de estos parámetros y realizar todas las combinaciones posibles, obteniendo así una lista de configuraciones para correr el algoritmo.

La idea detras de todas estas pruebas, es llegar a una configuración óptima (o cercana a la óptima en relación a estos casos) y poder aplicarla a nuevos casos de prueba y comparar con el valor óptimo. Esto significa que lo que realmente estamos haciendo en esta etapa es entrenar a nuestro algoritmo.

Los valores elegidos para realizar la experimentación son los siguientes:

4.1. Probabilidades de mutación

Las probabilidades de mutación con las que estaremos trabajando van a ser de 0.25, 0.5 y 0.75. En teoría cuanto más grande es esta probabilidad nuestro algoritmo más se parece a una elección random, con lo cual hipotéticamente deberíamos obtener mejores resultados utilizando la probabilidad de 0.25. Más adelante tendremos la oportunidad de comprobar si esto se da así.

4.2. Cantidad de iteraciones

La cantidad máxima de iteraciones va a ser 100, 1000 y 10000. Deberíamos tener mejores resultados con una cantidad mayor de iteraciones a menos que hayamos encontrado un máximo muy pronto, con lo cual el valor de 10000 debería ser el que exponga un mejor comportamiento del algoritmo.

4.3. Tamaño de la población

Los tamaños de la población van a ser 100, 500 y 1000. En este caso también podemos creer que cuanto más grande es este valor mejor se va a comportar el algoritmo porque vamos a tener una población más diversa. Sin embargo, podría suceder que los genotipos generados sean un tanto similares, con lo cual si bien vamos a tener una población grande en tamaño, no va a ser muy diversa. Por eso es que no utilizamos valores más grandes que 1000.

4.4. Matrices de entrenamiento

Corrimos el algoritmo con cada una de estas configuraciones sobre 5 matrices distintas: N-r100a2 (100x100), N-stabu3_250 (250x250), N-t1d500.10 (500x500), N-t1d500.25 (250x250), N-t65f11xx_250 (250x250)

4.5. Resultados

A continuación mostraremos los mejores resultados con los datos de configuración correspondientes y los tiempos de corrida del algoritmo. Compararemos por crossover, ya que este fue uno de los criterios con el que obtuvimos mayores diferencias.

4.5.1. Entrenamiento

Tabla 1: Resultados con crossover normal

	N-r100a2	N-stabu3_250	N-t65f11xx_250	N-t1d500.10	N-t1d500.25
Fitness value	134428	10342616	7310468	2206757	2210448
Tiempo (segundos)	1.429	4.351	8.897	17.249	45.712
Parent selection	Roulette	Roulette	Roulette	Roulette	Roulette
Probabilidad de mutación	0.75	0.25	0.25	0.75	0.75
Cantidad de iteraciones	10000	10000	10000	10000	10000
Tamaño de la población	100	100	100	100	100

Tabla 2: Resultados con variación en crossover

	N-r100a2	N-stabu3_250	N-t65f11xx_250	N-t1d500.10	N-t1d500.25
Fitness value	126016	9892540	6983509	2185039	2188596
Tiempo (segundos)	0.6	3.96	9	16.74	17.72
Parent selection	Roulette	Roulette	Roulette	Roulette	Roulette
Probabilidad de mutación	0.75	0.75	0.25	0.75	0.5
Cantidad de iteraciones	10000	10000	10000	10000	10000
Tamaño de la población	100	100	100	100	100

De las tablas anteriores podemos extraer algunas conclusiones. Lo más relevante de las tablas es que en todos los casos obtuvimos el mismo tamaño de población y cantidad máxima de iteraciones, 100 y 10000 respectivamente. Esto prueba un poco la hipótesis que habíamos definido más arriba, en la cual dijimos que tener una cantidad de iteraciones grande iba a hacer que mejore el comportamiento de nuestro algoritmo pero que sin embargo, esto podía no suceder con el tamaño de la población. Vemos que efectivamente el tamaño de población que terminó prevaleciendo es el más chico, lo cual significa que con 100 elementos en la población el algoritmo es capaz de generar nuevos miembros lo suficientemente variables para acercarse bastante al óptimo (esto se va a ver en el gráfico más abajo).

Otro parámetro que se mantiene es el tipo de selección de padres, que en todos los casos dió mejor utilizando roulette. Uno de los motivos por los cuales puede darse esto es por la manera en la que la selección por roulette tiene en cuenta los $fitness\ values$ de los elementos ya que cuando se utiliza la selección por tournament, los k elementos que se seleccionen en una primera instancia podrían tener $fitness\ values$ muy bajos. En cambio roulette hace una sola selección donde tiene en cuenta todos los elementos y sus $fitness\ values$.

En cuanto a la probabilidad de mutación, los resultados no se corresponden mucho con la hipótesis de que cuanto mayor era la probabilidad de mutación el algoritmo se va a comportar peor por su aleatoriedad. Si bien existen algunos casos donde la probabilidad de mutación de 0.25 prevaleció, los datos indican que deberíamos utilizar el valor de 0.75 para la experimentación final.

Finalmente se puede observar que hay una diferencia muy visible en los tiempos, que se manifesta especialmente en la última matriz de la tabla (N-t1d500.25). La razón por la cual sucedió esto es porque no se corrieron todos los experimentos en una misma pc si no que se utilizaron dos distintas. Más aún, el hardware de una tiene un procesador con dos nucleos más que la otra y esto afectó al tiempo de corrida del algoritmo. Más abajo damos la especificación de los hardware de ambas pc.

A continuación vamos a mostrar un gráfico de barras que compara los mejores resultados ($fitness\ values$ obtenidos) del algoritmo con un crossover y el otro y también agregamos información sobre donde se encuentra el valor óptimo. Dado que salvo en la matriz N-r100a2, en todos los demás se desconoce el valor óptimo, lo que hicimos fue proveer dos barras más (Mín óptimo y Máx óptimo, una al final y otra al principio) que denotan los extremos del intervalo donde se encuentra este máximo, que son valores que sí se conocen.

En el gráfico vamos a poder ver que el crossover standard termina siendo ligeramente superior al otro en todos los casos. Esto puede ser un indicador de que debemos respetar el orden absoluto de los elementos que el hijo hereda del primer padre, lo cual no pasa con el crossover variable.

normal vs variado.png

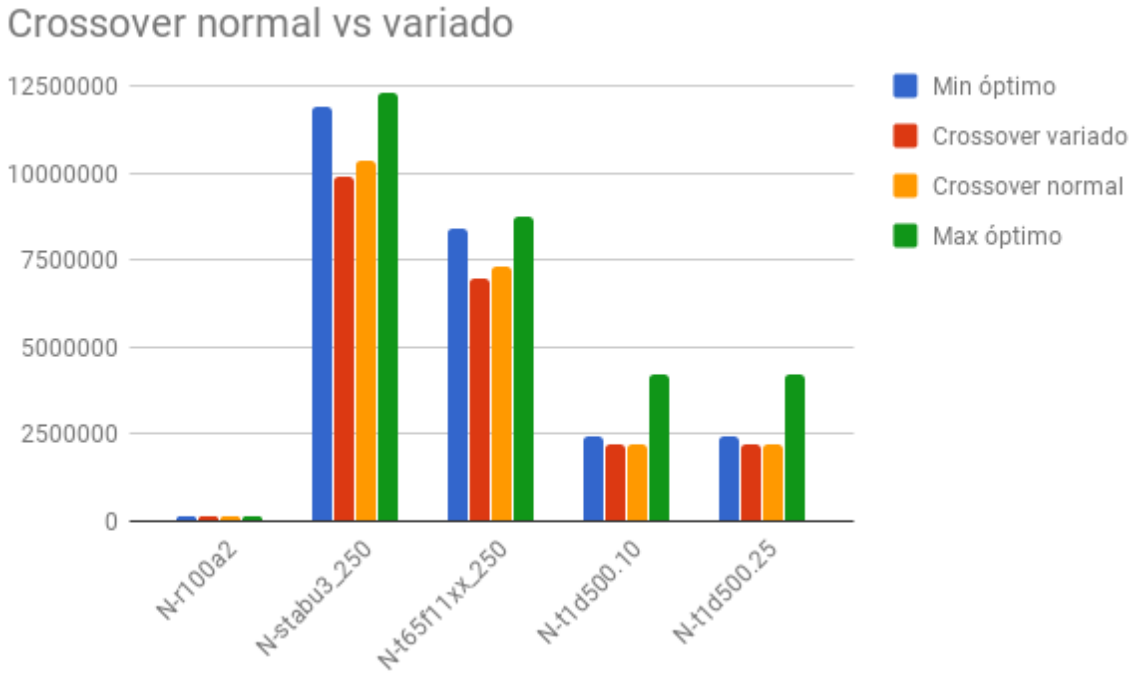

Figura 1

4.5.2. Resultados finales

A partir del análisis de los resultados mostrados anteriormente, decidimos correr nuestro algoritmo sobre tres nuevas matrices N-t1d100.05 (100x100), N-t1d500.20 (500x500) y N-tiw56r66_250 (250x250) utilizando la combinación de criterios y valores que nos arrojaron mejores resultados (roulette para selección de padres, 0.75 probabilidad de mutación, tamaño de la población de 100 y 10000 iteraciones).

Tabla 3: Resultados con variación en crossover

	N-t1d100.05	N-t1d500.20	N-tiw56r66_250
Fitness value	100270	2213413	4227781
Tiempo (segundos)	0.781256	17.54	5.407496
Parent selection	Roulette	Roulette	Roulette
Probabilidad de mutación	0.75	0.75	0.75
Cantidad de iteraciones	10000	10000	10000
Tamaño de la población	100	100	100

Como se desconoce la permutación óptima para estas matrices, el siguiente grafico compara los resultados obtenidos, utilizando nuestra mejor configuración, con el rango de valores donde se encuentra el óptimo.

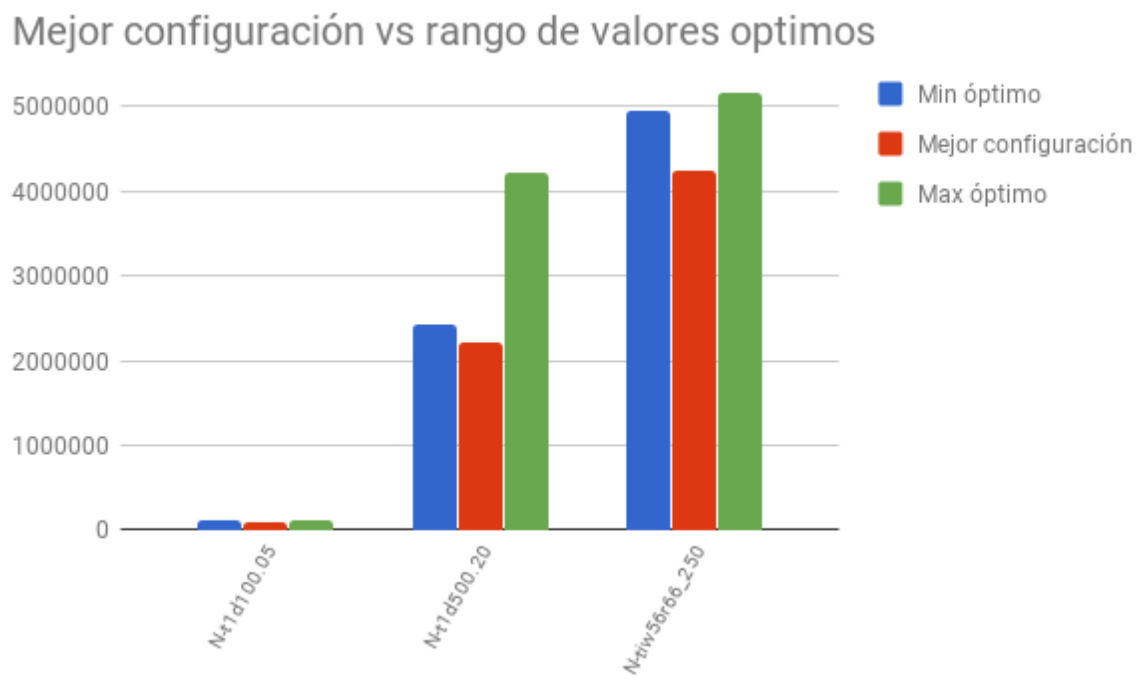

Figura 2

Podemos ver que los valores arrojados por el algoritmo están por debajo de la cota inferior conocida del valor óptimo, cosa que también se observa en el gráfico que muestra los resultados de la experimentación de entrenamiento. En una experimentación futura se podrían alterar los valores de cantidad de iteraciones y probabilidad de mutación, aumentándolos ya que el hecho de que los demás valores se hayan mantenido iguales en todas las combinaciones ganadoras nos hace pensar que no tendría mucho sentido alterarlos. Utilizar otros criterios de selección de padres y sobrevivientes también podrían ayudar a que el algoritmo tenga un mejor comportamiento.

4.5.3. Especificaciones de hardware/software

Como aclaramos más arriba, el algoritmo fue corrido en dos PC distintas. Ambas poseen 16 gb de RAM pero una tiene MacOS y un procesador de 5 nucleos, mientras que la otra utiliza Linux y un procesador de 7 nucleos. Cabe aclarar igual que la experimentación final se corrió enteramente en la pc de 7 nucleos.

5. Conclusiones

El presente trabajo nos permitió tener una primera experiencia en lo que es la implementación es un algoritmo genético, entendiendo el esquema de pasos que debe respetarse y aprendiendo sobre los distintos criterios que pueden aplicarse. Por medio de la experimentaicón pudimos ver que criterios de selección de padres funcionan mejor para el set de casos que elegimos.

Planteamos hipótesis que pudimos contrastar con la experimentación que hicimos, que resultó ser bastante extensa por la cantidad de configuraciones posibles. Además al plantear un set de casos de entrenamiento y uno de pruebas pudimos ver que tan bien funcionó la configuración ganadora para casos nuevos. Dicha configuración vimos que resultó ser con una probabilidad de mutación relativamente alta, cosa que no esperabamos, la selección de padres de tipo roulette, la cantidad más alta de iteraciones propuesta que fue 10000 y una población de 100 elementos.

Como trabajos futuros en cuanto a la implementación sería conveniente aprovechar la extensibilidad del código para agregar nuevos criterios de selección de padres. Por otro lado sería bueno modificar el esquema del algoritmo para que se puedan utilizar otros criterios de parada y experimentar con ellos. Finalmente en cuanto a la experimentación, dado que los valores ganadores de la cantidad de iteraciones y probabilidad de mutación fueron los máximos, se podrían agregar nuevos experimentos con valores aún más altos para ver cuanto más conviene aumentarlos, poniendo en la balanza el tiempo de cómputo y el *fitness value* obtenido.