

**TRABAJO DE LA ASIGNATURA DE INTERLIGENCIA ARTIFICIAL 1
PRESENTADO POR ANDREA CIMMINO ARRIAGA**

INDICE

0.0 – Introducción.....	1
1.0 – Planteamiento del problema.....	2
2.0 – Descripción del trabajo realizado.....	3
2.1 – Representación de los grafos.....	3
2.2 – Estructura de ficheros.....	5
2.3 – Trabajo realizado.....	6
2.4 – Objetivos del trabajo.....	7
3.0 – Algoritmo Genético Híbrido.....	8
3.1 – Algoritmo Genético.....	9
3.1.1 – Representación del problema.....	9
3.1.2 – Pseudocódigo.....	10
3.2 – Primera implementación.....	12
3.2.1 – Funciones implementadas.....	12
3.2.2 – Información específica.....	13
3.3 – Segunda Implementación.....	14
3.3.1 – Mejoras.....	14
3.3.2 – Funciones implementadas.....	14
3.3.3 – Información específica.....	15
3.4 – Tercera Implementación.....	15
3.4.1 – Mejoras.....	15
3.4.2 – Funciones implementadas.....	16
3.4.3 – Información específica.....	16
3.5 – Algoritmo de Enjambre.....	16
3.5.1 – Pseudocódigo.....	16
3.5.2 – Funciones Implementadas.....	17
3.5.3 – Información específica.....	18
4.0 – Algoritmos Genéticos Paralelos.....	19
4.1 – ¿Cómo funciona el modelo de islas?.....	19
4.2 – Representación del problema.....	20
4.3 – Pseudocódigo.....	20
4.4 – Funciones implementadas.....	21
4.5 – Información específica.....	23
5.0 – <i>Benckmarks</i>	25
5.1 – Grafos usados en el trabajo.....	25
5.2 – Grafo Pentágono.....	26
5.3 – Grafo de los puentes de Königsberg.....	28
5.4 – Grafo de Petersen.....	30
5.5 – Grafo del mapa de Andalucía.....	32
5.6 – Grafo Bipartito.....	34
5.7 – Problemas de optimización que no son de coloración.....	35
6.0 – Conclusiones.....	36
7.0 – Bibliografía y webgrafía.....	37

0.0 – Introducción.

El presente trabajo tiene como objeto implementar unos algoritmos para resolver el problema de coloreado de grafos.

Mi elección fue basar mi entrega en los algoritmos genéticos, porque me fascina su modularidad, gracias a las funciones que los componen. Precisamente esta característica les da mucho juego y potencia, pudiendo obtener muchos algoritmos distintos, modificando ligeramente alguno de esos módulos. **Mi primera elección** fue la del **genético híbrido, descrito en el trabajo de Musa M. Hindi y Roman V. Yampolskiy**, que está colgado en la web. **Mi segunda elección fue el modelo de islas**, me interesó sobre manera debido a la simplicidad de la idea, y al mismo tiempo a la potencia que aporta.

En el algoritmo híbrido genético he realizado varias implementaciones, porque me pareció muy interesante ver cómo mejoraba su eficiencia, en base a como se iba adaptando al estilo de programación funcional (estructuras de datos, uso de funciones nativas, etc) de LISP. Haber hecho tres versiones distintas da una visión de una evolución real, que es el proceso que realicé a la hora de completar el trabajo. Podría haber incluido sólo la versión final, pero me pareció adaptó e interesante ver cómo puede afectar el uso de unas estructuras de datos y el uso de funciones nativas a la eficiencia.

Por otro lado, en el modelo de islas podría haber implementado un algoritmo genético más simple, pero realmente como ya tenía bastante trabajo y funciones hechas de la primera parte, me pareció más interesante jugar con el algoritmo de Musa M. Hindi y Roman V. Yampolskiy. Eso me llevó a rediseñar, en parte, su algoritmo adaptándolo al modelo de islas, pero quedándome con las partes que me parecían más eficientes y mejores.

El resultado de todo esto configura el presente trabajo.

1.0 – Planteamiento del problema.

El **problema que se plantea en este trabajo** es un clásico problema de la inteligencia artificial, denominado **coloreado de grafos**. Se trata de un caso especial de etiquetado de grafos si se tratan los colores como números o letras. Normalmente lo que se colorean son los nodos de los grafos (**vértice coloración**), aunque existe la posibilidad de colorear las aristas (arista coloración). En este trabajo nos quedaremos con la coloración de los vértices, que se lleva estudiando desde aproximadamente 1852, cuando Francis Guthrie¹ postuló la conjetura de los cuatro colores, según la cual es suficiente ese número para pintar cualquier mapa, hasta nuestros días.

La estructura de datos que se nos plantea es un conjunto de nodos unidos por aristas, y **se nos pide que le asignemos un color o etiqueta a cada vértice, cumpliéndose en todo momento la condición de que ningún vértice tendrá o se le asignará un color que ya tenga otro de sus nodos vecinos (con los que está unido).**

Computacionalmente, este problema es de tipo NP-completo², que pertenece a la lista de Karp³. Para tratar de resolverlo, existen múltiples tipos de algoritmos y estrategias.

1 http://es.wikipedia.org/wiki/Francis_Guthrie

2 <http://es.wikipedia.org/wiki/NP-completo>

3 http://es.wikipedia.org/wiki/Lista_de_21_problemas_NP-completos_de_Karp

2.0 – Descripción del trabajo realizado.

En este trabajo se abordan dos problemas principalmente, el primero es la coloración de grafos (descrito anteriormente), y el segundo es algo intrínseco a las ciencias de la computación.

Con respecto a la coloración de grafos, se parte de dos algoritmos, el primero está sacado del artículo, recomendado en la web del departamento para este trabajo, “*Genetic Algorithm Applied to the Graph Coloring Problems*” de Musa M. Hindi y Roman V. Yampolskiy. El segundo algoritmo consiste más bien en una estrategia de abordaje del problema, denominado modelo de islas, el cual se combina con un algoritmo genético cualquiera.

En segundo lugar, como mencioné anteriormente, en la realización de este trabajo surge un segundo problema: todo el pseudocódigo de los artículos que he leído y el planteamiento funcional de las implementaciones sigue una filosofía de programación declarativa, mientras que este trabajo se ha realizado todo en LISP, que pertenece a la rama de la programación funcional. Con lo cual las estructuras de datos, funciones y otros elementos han tenido que ser adaptados o rediseñados, porque en otro caso (como se vera más adelante), los tiempos de ejecución serán mucho mayores que los obtenidos en los artículos.

2.1 – Representación de los grafos.

Por otro lado también existe otro asunto relacionado con la computación, que es cómo representar los grafos y los colores. Para poner a prueba los algoritmos desarrollados (y como se puede ver en el apartado de *Benchmarks*), he usado un grafo presente en la web <http://www.info.univ-angers.fr/pub/porumbel/graphics/> entre otros. Los grafos contenidos en los archivos de esa web siguen un esquema fijo, cada línea puede empezar por una de estas tres letras:

- c : si la línea comienza por “c”, significa que es un comentario
- p : si la línea empieza por “p”, contendrá un espacio en blanco, el número de nodos y el número de aristas.
- e : contendrá dos números que corresponden a dos vértices y representa que están unidos.

En principio, se podría haber usado esta estructura para representar los grafos, pero puesto que en el artículo de “*Genetic Algorithm Applied to the Graph Coloring Problems*” (que fue el primero sobre el que trabajé), se habla del uso de una matriz de adyacencia para representar los grafos, al final me decanté por usar esta última estructura.

Para adaptar los ficheros de *benchmarks* a la estructura elegida por mí, y necesaria en los algoritmos, he desarrollado un pequeño script en python, que se encuentra presente en todas las carpetas que contienen los algoritmos. Se denomina “**graph_converter.py**”. Dicho programa se invoca pasándole como parámetros el nombre del fichero a convertir (por ejemplo: *graph_andalucia.txt*) y el número de colores con los que se desea colorear el mapa. Entonces se generará un archivo *data.lsp* que contiene el número de vértices del grafo (v), el número de colores (k), y la matriz de adyacencia.

Veamos este proceso con capturas de pantalla:

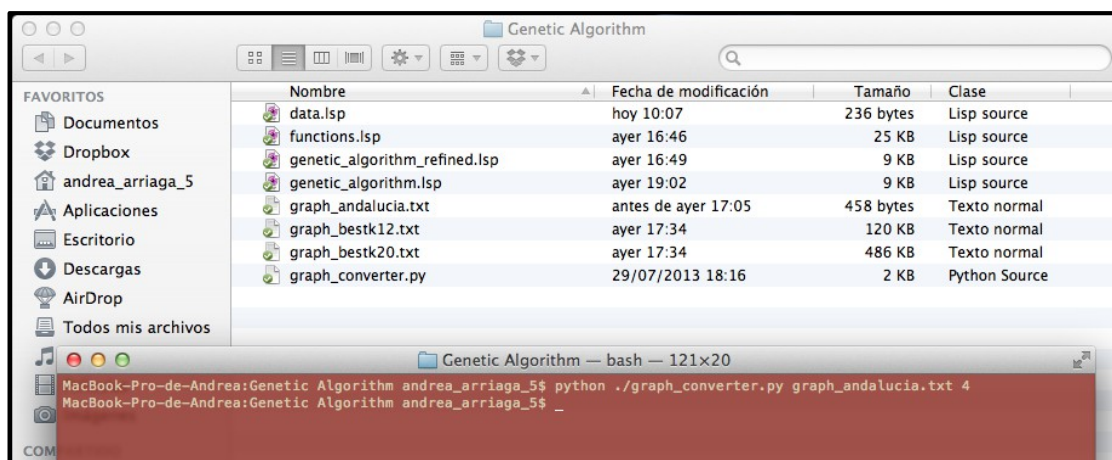
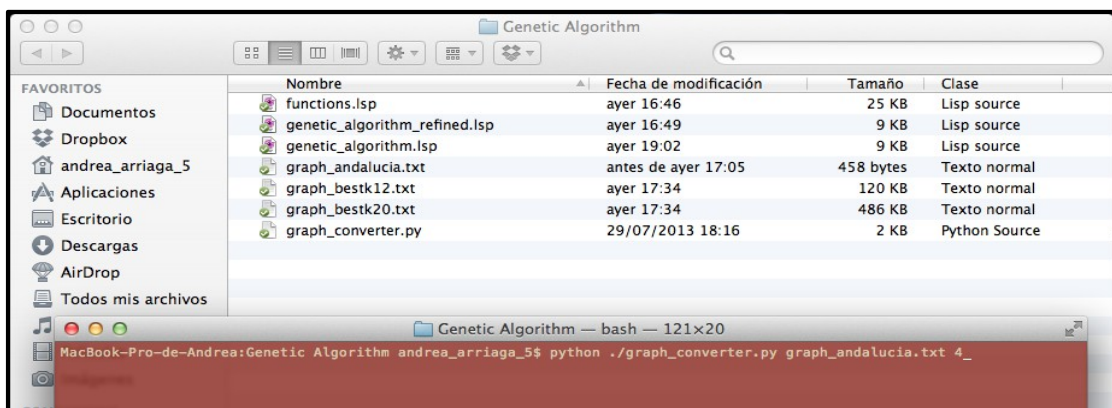
- Fichero graph_andalucia.txt

```

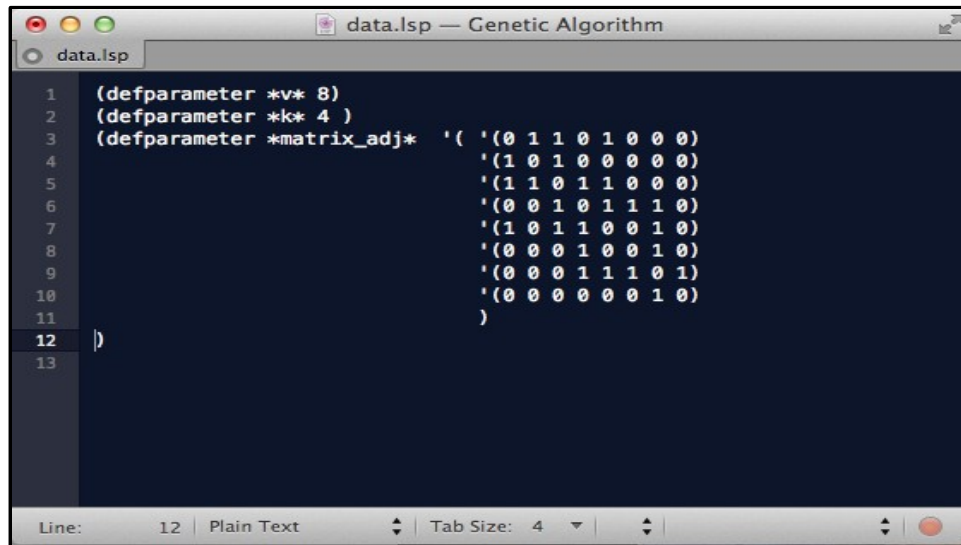
c File: graph_andalucia.txt
c
c SOURCE: Andrea Cimmino Arriaga
c DESCRIPTION: It's the andaluci's map translated to a graph.
c Nodes description:
c
c   - CADIZ: 1
c   - HUELVA: 2
c   - SEVILLA: 3
c   - CORDOBA: 4
c   - MALAGA: 5
c   - JAEN: 6
c   - GRANADA: 7
c   - ALMERIA: 8
c
p edge 8 246788
e 1 2
e 1 3
e 1 5
e 2 1
e 2 3
e 3 1
e 3 2
e 3 4
e 3 5
e 4 3
e 4 5
e 4 6
e 4 7
e 5 1
e 5 3
e 5 4
e 5 7
e 6 4
e 6 7
e 7 4
e 7 5
e 7 6
e 7 8
e 8 7

```

- Invocando al programa *graph_converter.py* (podemos observar que no existe ningún fichero en el directorio denominado “data.lsp”, si existiera no ocurriría nada, simplemente se sobrescribiría con los datos nuevos).



- Fichero que se ha creado:



```
1 (defparameter *v* 8)
2 (defparameter *k* 4)
3 (defparameter *matrix_adj* '( (0 1 1 0 1 0 0 0)
4                               '(1 0 1 0 0 0 0 0)
5                               '(1 1 0 1 1 0 0 0)
6                               '(0 0 1 0 1 1 1 0)
7                               '(1 0 1 1 0 0 1 0)
8                               '(0 0 0 1 0 0 1 0)
9                               '(0 0 0 1 1 1 0 1)
10                              '(0 0 0 0 0 0 1 0)
11                              )
12 )
13
```

Resumiendo, **para la representación del problema de coloreado almaceno** (como se puede ver en la captura anterior), **una matriz de adyacencia para el grafo**, el **número de vértices** (el número que aparece es el número total de vértices, en este caso 8) **y el número de colores de los que disponemos para colorear** (el color 0 no existe). Estos datos serán los que usaran todos los algoritmos para colorear los grafos.

2.2 – Estructura de ficheros.

El trabajo se compone de este documento y varias carpetas:

- HGenetic Algorithm.
- HGenetic Algorithm refined.
- Islands Model.

Todas las carpetas siguen la misma estructura de ficheros:

- “*data.lsp*”: **contendrá el grafo** que se desea colorear (representado por su matriz de adyacencia), el **número de vértices y el número de colores** con los que se desea colorear el grafo.
- “*functions.lsp*”: **contiene una serie de funciones** que se usarán luego en el fichero, que se llama igual que la carpeta.
- Los **ficheros que se llaman igual que la carpetas** en los que están contenidos **tienen definidos los parámetros de cada algoritmo**, esto es, todo aquello que necesitará cada algoritmo para encontrar la solución. Además **también tendrá el código principal del algoritmo que sea**. En la primera carpeta, HGenetic Algorithm, existen dos ficheros de algoritmo principal. El primero se llama igual que la carpeta y el segundo (que es una implementación mejorada de este primero) se denomina “*genetic_algorithm_improved.lsp*”.

- “*graph_converter.py*”: es una utilidad desarrollada por mí de la cual ya se ha hablado.
- Varios ficheros “.txt” que contienen distintos grafos, se hablará de ellos en la sección de *Benchmarks*.

2.3 – Trabajo realizado

Lo que presento en este trabajo son tres implementaciones del algoritmo genético híbrido descrito en el artículo recomendado por el departamento “*Genetic Algorithm Applied to the graph Coloring Problem*”. Aunque las tres son el mismo programa varían mucho una de otra, como se expondrá más adelante. Esas grandes diferencias surgen debido a que en el artículo todo está enfocado desde un punto de vista de la programación declarativa y al implementar todo tal cual en programación funcional la eficiencia baja drásticamente, así que las dos siguientes implementaciones tratan de mejorar los tiempos de ejecución.

Por otro lado, además de lo anterior he implementado un modelo denominado **modelo de islas**, que no es exactamente un algoritmo. Consiste en una estrategia para subdividir un problema en partes y lanzar cada una con un algoritmo genético. Como de esta parte lo que me pareció más interesante no fue en sí el algoritmo genético, sino el paralelismo o la simplificación del problema, el genético que uso en el modelo es de lo más simple, construido a partir del genético usado para la primera parte de este proyecto.

Por lo tanto, **resumiendo**, tenemos:

- **Genético Híbrido primera implementación** (corresponde al fichero “*hgenetic_algorithm.lsp*”, contenido en la carpeta HGenetic Algorithm. Sólo es el algoritmo genético, el de enjambre no está implementado debido a la ineficiencia computacional de las estructuras de datos).
- **Genético Híbrido segunda implementación** (corresponde al fichero “*hgenetic_algorithm_improved.lsp*”, contenido en la carpeta HGenetic Algorithm. Sólo es el algoritmo genético, el de enjambre no está implementado, debido a la ineficiencia computacional de las estructuras de datos).
- **Genético Híbrido tercera implementación** (corresponde al fichero “*hgenetic_algorithm_refined.lsp*”, contenido en la carpeta Hgenetic Algorithm Refined. Tiene implementado también el algoritmo de enjambre).
- **Modelo de Islas con Genético Simple** (corresponde al fichero “*islands_model.lsp*”, contenido en la carpeta Islands Model).

Además de todo esto, el presente trabajo que contiene los resultados de haber lanzado todos esos algoritmos sobre una serie de grafos (sección *Benchmarks*), y haber recopilado sus tiempos de computación y haberlos comparado.

2.4 – Objetivos del trabajo

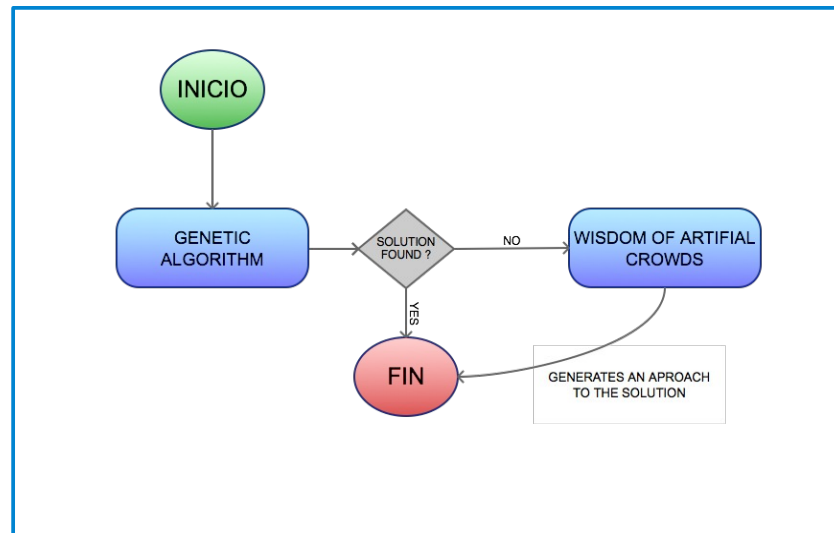
Los objetivos del presente trabajo son implementar los algoritmos descritos en el apartado anterior y obtener unos tiempos “aceptables” de ejecución (mostrados en el apartado de *Benchmarks*). Además de eso ,cumplir los requisitos descritos en la página del departamento, que son:

Objetivos descritos en el departamento	Apartados del trabajo
Estudiar y comparar distintos algoritmos de optimización, utilizados para resolver el problema planteado.	He implementado dos algoritmos distintos (genético híbrido y modelo de islas) y comparado los tiempos de ejecución para una serie de problemas (ilustrado en el apartado de <i>Benckmarks</i>)
Para cada uno de los algoritmos indicar los elementos necesarios para representar un problema de optimización cualquiera.	La sección 2.1 <i>Representación de grafos</i> describe como representar los grafos en los problemas y respectivamente cada sección de algoritmo tiene un subapartado indicando las estructuras de datos que usan para representar el problema.
Para cada uno de los algoritmos incluir un pseudocódigo que describa su funcionamiento.	Cada algoritmo tiene un subapartado denominado pseudocódigo, conteniendo ésta información.
Para cada uno de los algoritmos incluir un par de ejemplos de problemas sencillos, su formalización y los resultados obtenidos (problemas distintos al de la coloración de grafos)	Este subapartado no se ha podido realizar por una serie de motivos que están explicados en la sección de <i>Benchmarks</i> .
Comentar los resultados obtenidos y justificar la elección de los algoritmos presentados.	Sección de <i>Benckmarks</i> y conclusiones.

3.0 – Algoritmo Genético Híbrido

Este apartado del trabajo está sacado del artículo “*Generic Algorithm Applied to the Graph Coloring Problem*” de Musa M. Hindi y Roman V. Yampolskiy en el que se presenta una técnica híbrida, que aplica un algoritmo genético seguido de otro algoritmo, conocido como “*Wisom of artifiacd Crowds*” (o algoritmo de enjambre en español) para resolver el problema de la coloración de grafos.

El flujo de ejecución que presentan Musa M. Hindi y Roman V. Yampolskiy es el siguiente:



Podemos observar que realmente **lo que se hace es aplicar el algoritmo genético**, del cual hablaremos en breve, **sobre un problema de coloración y en caso de no encontrar una solución, entonces se lanza el algoritmo de enjambre**.

Podemos dividir, por lo tanto, el contenido del artículo en dos primeras partes: el algoritmo genético y el algoritmo de enjambre. En cada sección se nos presenta el pseudocódigo y una breve descripción de las funciones que implementan cada algoritmo. En la parte final del *paper*, se muestran unos resultados obtenidos, tras lanzar este genético híbrido sobre una serie de grafos.

Antes de pasar al análisis de cada parte, me gustaría mencionar que **el artículo en sí (y sus algoritmos) fue desarrollado pensando en un estilo de programación declarativa y usando como lenguaje JAVA, mas concretamente JDK y un *framework* denominado JUNG**. Por lo tanto todo, el pseudocódigo que expondré está orientado a esa filosofía, aunque el trabajo se haya realizado en programación funcional usando LISP. Esta diferencia me ha llevado a desarrollar tres implementaciones del algoritmo del *paper*: el primero es una traducción literal, el resultado es una eficiencia pésima, ya que se llama muchas veces a una función (de la cual hablaré mas adelante), que resulta ser el cuello de botella del algoritmo, muchísimas veces. El motivo de que sea tan lenta es porque tiene que recorrer una lista enorme de datos y realizar operaciones de inserción y búsqueda. Este gran problema sólo se presenta al programar en LISP (y para problemas grandes), ya que en JAVA y, usando un *framework*, los tiempos acceso son considerablemente menores que los conseguidos aquí. De este problema surge la primera mejora que he diseñado, la cual simplemente (luego ilustrare cómo) invoca a la función cuello de botella una vez por iteración, reduciendo drásticamente el tiempo de ejecución. Desgraciadamente esta mejora aún no iguala los tiempos conseguidos por Musa M. Hindi y Roman V. Yampolskiy con su implementación. Para alcanzar la misma eficiencia que ellos, rediseñé en parte el algoritmo, o más bien, sus estructuras de datos de

tal forma que se reducen las operaciones por iteración y, además, se saca provecho de las funciones nativas de LISP, lo que mejora muy notablemente los tiempos. Con esta última mejora consigo alcanzar unos tiempos muy rápidos (como se puede apreciar en la última sección, donde expongo los tiempos de ejecución de cada algoritmo para un mismo problema).

3.1 – Algoritmo Genético

En el artículo se nos presenta un algoritmo genético modificado. **El pseudocódigo en principio no varía mucho de un genético normal:**

```
population = randomly generated chromosomes;

while( terminating conditions is not reached){
    gaRun();
}

//an iteration of a genetic algorithm
Function gaRun(){
    parents = getParents();
    child = crossover( parents );
    child = mutate( child );
    population.add( child );
}
```

Lo que se pretende en el *paper* es conseguir mejorar la “bondad” (*fitness*) de la población (los cromosomas). Para ello se van a aparear (*crossover*) a dos de los individuos más aptos (*parents*), para obtener en cada iteración una población superior a la anterior, es decir, que vaya convergiendo a la solución. Cada iteración que se realice generará una denominada generación. Aunque se generen nuevos individuos, el tamaño de la población no variará y se mantendrá constante a 50 (establecido por los creadores). **Este proceso finalizará al alcanzar la condición de parada**, Musa M. Hindi y Roman V. Yampolskiy establecen para este algoritmo dos condiciones de parada: **o se encuentra una solución o se alcanza un número de generación máximo**, preestablecido, en este caso en 20.000.

3.1.1 – Representación de datos

La representación de datos que se presenta es la siguiente:

- **Un cromosoma** es una array, cada una de sus posiciones representa un vértice y el contenido de cada posición (un entero) un color.
- **La población** es una lista de cromosomas.
- **La generación** y el **número máximo de generación** son simples contadores (enteros).
- **El grafo** sobre el que se trabaja es una matriz de adyacencia.
- Además se usan dos enteros, uno para representar el **número de nodos** y otro los **colores disponibles**.

3.1.2 - Pseudocódigo

La innovación que se presenta en el artículo reside en las funciones que escogen a los padres, los cruzan y mutan a los hijos. En el *paper* se implementan dos funciones para lo primero (*parentSelection1* y *parentSelection2*) una para el cruce (*crossover*) y dos para lo último (*mutation1* y *mutation2*).

ParentSelection1

```
tempParents = two randomly selected chromosomes  
from the population;  
parent1 = fitter of tempParents;  
  
tempParents = two randomly selected chromosomes  
from the population;  
parent2 = fitter of tempParents;  
  
return parent1, parent2;
```

Lo que hace la función es escoger aleatoriamente dos individuos de la población y se queda con el mejor (menor fitness), ese será el primer padre. Para obtener el segundo realiza el mismo proceso.

ParentSelection2

```
parent1 = the top performing chromosome;  
parent2 = the top performing chromosome;  
  
return parent1, parent2;
```

En este caso los padres son los dos cromosomas de la población con menor fitness.

Crossover

```
crosspoint = random point along a chromosome;  
child = colors up to and including crosspoint from  
parent 1 + colors after crosspoint in the end of the  
chromosome from parent2;  
  
return child;
```

Se cruzan los dos padres, escogiendo una posición aleatoria del cromosoma y cogiendo la primera parte hasta dicha posición de *parent1* y uniéndola a la parte obtenida desde ese punto hasta el final de *parent2*.

Por ejemplo:

```
Crosspoint = 2  
parent1 = ( 0 1 2 3 )  
parent2 = ( 4 5 6 7 )  
child = ( 0 1 2 ) + ( 4 )  
resultado = ( 0 1 2 4 )
```

Mutation1

```
for each (vertex in chromosome){
    if ( vertex has the same color as an adjacent vertex ){
        adjacentColors = all adjacent colors;
        validColors = allColors – adjacentColors;

        newColor = random color from validColors;
        chromosome.setColor(vertex, Newcolor);
    }
}
return chromosome;
```

Lo que se hace en la *mutation1* es a cada vértice con un color conflictivo se le asigna otro de entre los que no generan conflicto.

Aunque los autores no contemplan el caso en que no haya colores a elegir, en este caso mi algoritmo deja el vértice con el mismo color.

Mutation2

```
for each (vertex in chromosome){
    if ( vertex has the same color as an adjacent vertex ){
        newColor = random color from allColors;
        chromosome.setColor(vertex, newColor);
    }
}
return chromosome;
```

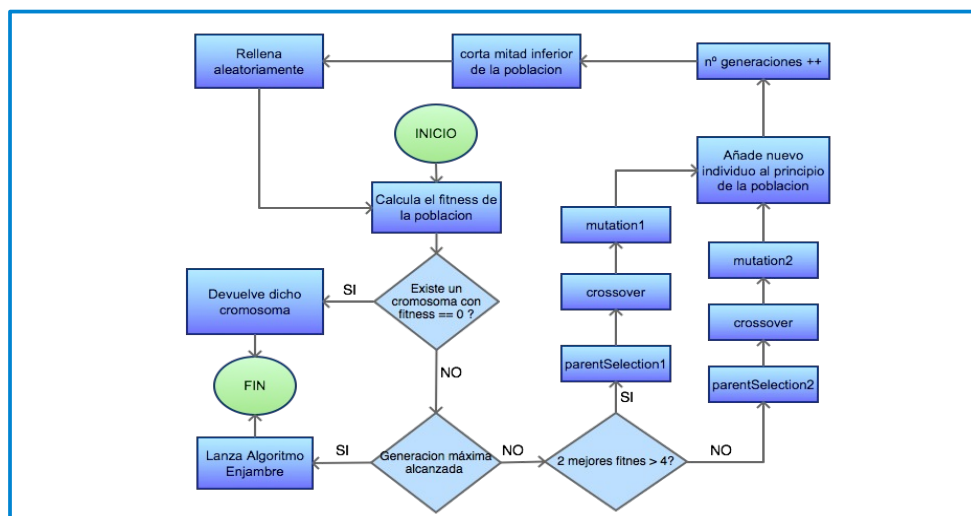
La *mutation2* simplemente le asigna un color aleatorio a un vértice conflictivo.

En cada iteración se calcula la bondad de la población, el *fitness* para cada cromosoma. Este proceso consiste en ver en cada cromosoma cuántos nodos están conectados con otros del mismo color. Sumando cada arista que una ese tipo de nodos se calcula el *fitness* de un vértice. Repitiendo este proceso para todos los nodos del grafo y un cromosoma se calcula la bondad del cromosoma.

Una vez que se tiene el *fitness* de todos los cromosomas se escogen los dos con menor puntuación y se ve si esta es mayor que 4, en tal caso se ejecuta el algoritmo usando las funciones de *parentselection1* y *mutation1*. En caso de que las dos puntuaciones sean menores o iguales que 4, entonces se usa *parentselection2* y *mutation2*.

Por lo tanto, se impone una restricción tácita, que la población nunca podrá ser menor que 2, ya que se requiere de este número de individuos para calcular qué tipo de funciones lanzar.

Para resumir un poco todo lo explicado tenemos el siguiente flujo de ejecución:



Si observamos el diagrama y el pseudocódigo en una iteración, primero tenemos que calcular quiénes son los dos mejores individuos de la población, teniendo que recorrer la lista y calculando el *fitness* de cada cromosoma. Toda esta operación tiene un gran coste computacional. Después, si resulta que los dos tienen un *fitness* menor que 4 en *parentselection2*, tendremos que volver a buscar dichos individuos (ya que en el pseudocódigo no se almacenan en ningún lado la primera vez). Además, en cada iteración hay que ver si se ha alcanzado una solución óptima. Para ello es necesario saber si existe un cromosoma con una puntuación de cero (sin conflictos), lo que nos lleva otra vez a buscar dichos individuos, es más, en esta parte, puede llegar a buscarla dos veces: una para saber si existe y devolver un valor *booleano*, y la segunda, para obtenerlos.

Con lo cual, el resultado es que llamaremos muchas veces a la misma función para obtener idénticos resultados. Teniendo en cuenta que la operación que realiza no es nada trivial, los tiempos de ejecución se disparan. Este es el gran cuello de botella del algoritmo (que en mi trabajo se llama *fitters_chromosomes* o *bests_chromosomes*). A la hora de intentar mejorar la implementación, este es el punto clave que hay que rediseñar. Como dije antes, en la primera versión se mantienen todas esas llamadas, en la segunda, tras realizarla la primera vez para saber si existe una solución, también la almaceno en un parámetro y, de esta manera, ya no hay que volver a pasar por el cuello de botella. Los tiempos mejoran considerablemente, como se puede apreciar en la última sección.

3.2 – Primera implementación.

La primera implementación es la menos eficiente de todas, debido a que realiza muchas veces la operación de calcular el *fitness* de la población y escoger el cromosoma con menor puntuación. Este problema, del que ya he hablado antes, es el que (como se refleja en el apartado de *Benchmarks*), lleva a una ineficiencia extrema en este caso.

La estructura de datos usada es la antes descrita, en el fichero “.lsp” corresponde a los siguientes parámetros:

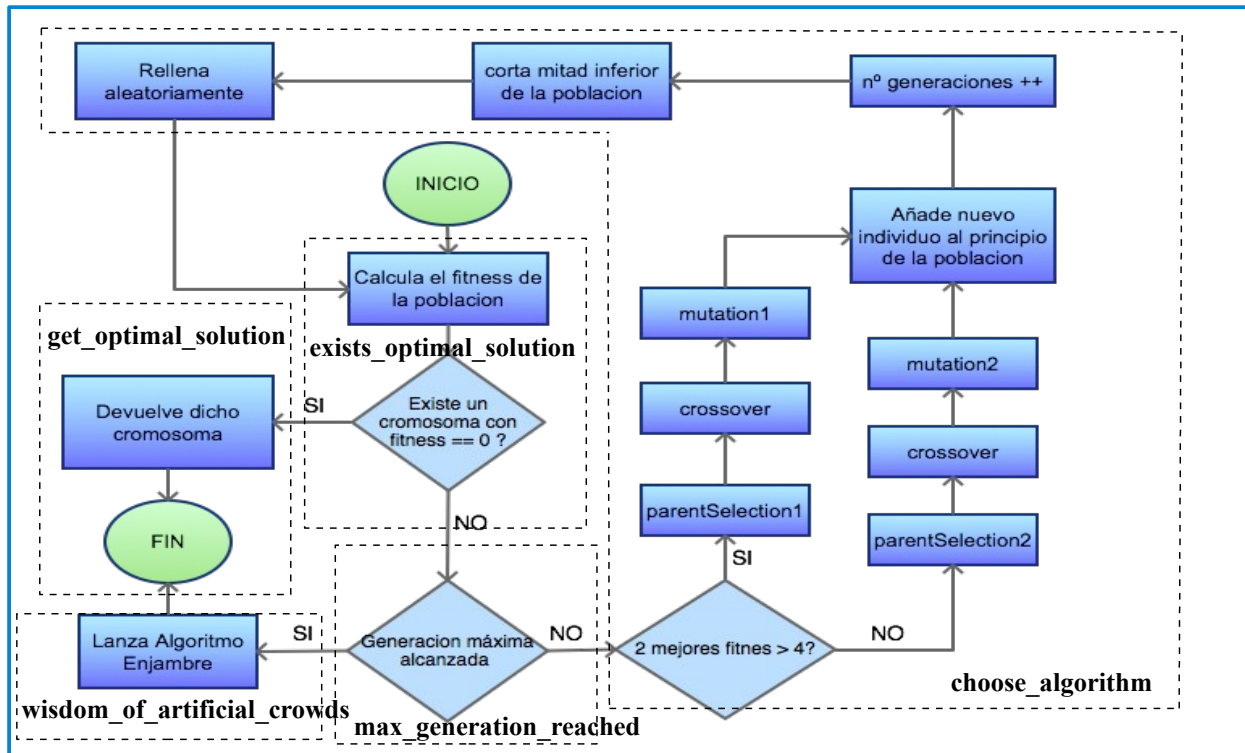
- ***population*** : Lista de cromosomas, se inicializa al lanzar la función *start* (ver más abajo) de forma aleatoria.
- ***max_generation*** : Entero.
- ***generation*** : Entero.
- ***population_length*** : Entero, es un parámetro auxiliar, que equivale al número de elementos de la lista.
- ***half_population_length*** : Entero, es un parámetro auxiliar, que equivale a la mitad número de elementos de la lista.

3.2.1 – Funciones implementadas

Las funciones principales que he escrito para cubrir la funcionalidad del algoritmo en esta primera versión son:

- *exists_optimal_solution*
- *get_optimal_solution*
- *max_generation_reached*
- *choose_algorithm*: engloba los *parentselection1* y *2*, *crossover*, *mutation1* y *2* y *create_new_generation*.
- *wisdom_of_artificial_crowds*
- *start* : engloba todas las funciones anteriores y lanza el algoritmo.

Como lo que realiza cada función está bastante claro con el nombre, además de que en los ficheros de código están claramente explicadas y detalladas, en este trabajo mostraré el flujo de ejecución del algoritmo y asignaré distintas funcionalidades a cada función de las anteriores. Lo que me gustaría señalar es que *exists_optimal_solution*, *get_optimal_solution*, *parentselection2* usan una función auxiliar que es *fitters_chromosomes*, la cual devuelve los dos cromosomas con menor *fitness*, presentes en la población. Esta función es la llamada cuello de botella y culpable (junto con la estructura de datos usada), de la ineficiencia de las dos primeras implementaciones.



3.2.2 – Información específica.

Realmente aquí indicare una serie de restricciones que cumple el algoritmo, algunas son en base a la implementación planteada por los autores, otras tácitas con la propia programación:

- Población constante a 50 cromosomas (generada aleatoriamente).
- Generaciones máximas 20.000.
- En la *mutation1* no se contempla que no haya colores disponibles para elegir. En dicho caso, yo he decidido dejar el color que tenía el vértice.
- Los cromosomas siempre tienen igual tamaño (como es lógico por otra parte).
- Si se varía la población lo mínimo son dos cromosomas.
- La población tiene que ser un número par, en cualquier caso.
- Debido a la mala eficiencia, esta versión no implementa el algoritmo de enjambre (*wisdom_of_artificial_crowds*). Si se llega a llamar a dicha función esta devolverá 0 para indicar que no se ha encontrado solución. La razón de esto es que el algoritmo tarda mucho tiempo en computar problemas y para que se lance esta función hay que llegar a 20.000 generaciones, lo cual puede llevar mucho tiempo en esta versión.

3.3 – Segunda Implementación.

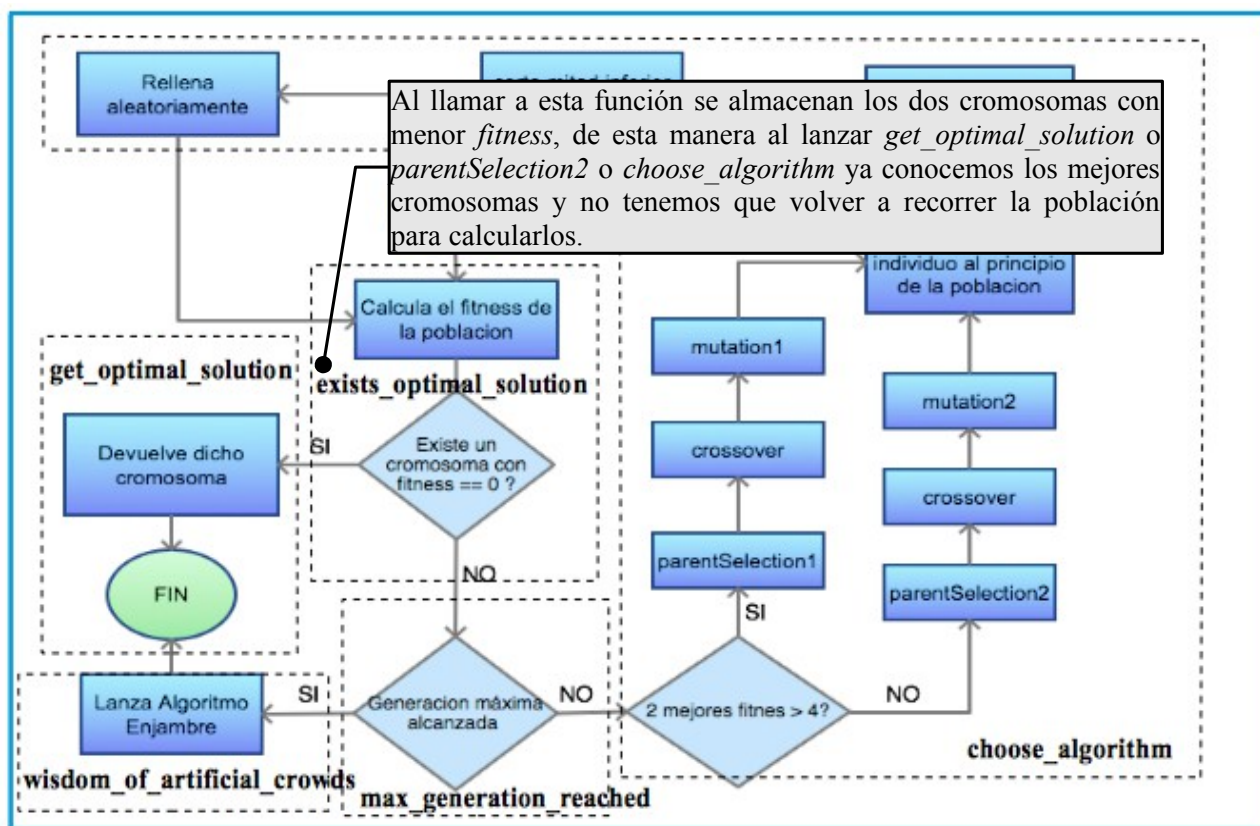
Esta segunda implementación se mejora considerablemente los tiempos de ejecución, aún así no cumple unos tiempos mínimos aceptables. Las estructuras de datos siguen siendo las mismas.

3.3.1 – Mejoras

La principal mejora de esta versión es que, aunque se mantengan las estructuras de datos de la versión anterior, **se reduce el número de veces que se llama a la función cuello de botella a una vez por iteración**. Como esta claro, esto hace que el tiempo de ejecución disminuya mucho. Aun así, sigue siendo muy poco. Para llevar a cabo esta mejora creo un parámetro que se denomina **best_chromosomes**, que es una lista con los dos mejores cromosomas de la población.

3.3.2 – Funciones implementadas

Las funciones implementadas en este apartado son las mismas que en el anterior, aun así veamos el flujo de ejecución (igual que el anterior), en el cual indicaré cómo se mejora la eficiencia.



Como vemos, al almacenar los mejores cromosomas en la primera llama de *exists_optimal_solution*, el resto de funciones que necesiten dicha información ya la tendrán almacenada en la variable **best_chromosomes**, por lo que no tendrán que iterar sobre la población y se ahorrará mucho tiempo.

3.3.3 – Información específica

Las restricciones de esta implementación son exactamente iguales que las de la primera, ya que como he indicado antes, lo único que cambia es el número de veces que se llama a la función cuello de botella.

3.4 – Tercera Implementación.

Esta versión es la implementación final del algoritmo genético híbrido (incluye también la función *wisdom_of_artificial_crowds* tal como se indica en el artículo). Sus tiempos de computación son muy buenos, gracias a las mejoras introducidas.

Como decía en la introducción, esta primera parte del trabajo fue una evolución constante del algoritmo genético para buscar la eficiencia máxima que LISP podía ofrecer. Las dos versiones anteriores fueron solo una introducción que dio pie a que esta última implementación viera la luz. Como veremos a continuación, éste algoritmo usará lo mejor de las versiones anteriores, pero evolucionara más allá que ellas para poder alcanzar la máxima eficiencia.

3.4.1 – Mejoras

El secreto de la eficiencia de esta versión reside en varios puntos clave: el primero es el **cambio de la estructura de la población**. Antes la población era una lista de cromosomas, **ahora es una A-Lista de pares cromosoma y puntuación de *fitness* asociada**. Esto ya ahorra el tener que calcular en cada iteración el *fitness*, porque si recordamos, solo la mitad de la población es sustituida por una nueva y aleatoria, por lo tanto la puntuación de cada cromosoma se calcula sólo para la mitad de la población por iteración. Esto ya es una mejora sustancial. Pero hay más, cuando se lanza por primera vez la función *exists_optimal_solution*, se almacenan los dos mejores pares *cromosoma-fitness* en una variable igual que en la segunda implementación. Con todo lo anterior ya hemos mejorado mucho la eficiencia, pero **la clave de esta implementación reside en una función nativa que explota una capacidad de las A-listas**. Recordemos que “el talón de Aquiles” de las demás implementaciones era la función *fitters_chromosomes*, que buscaba (calculando la puntuación de cada cromosoma), los dos mejores individuos. **Gracias a la nueva estructura de datos, ahora en lugar de usar esa, utilizo otra denominada *best_chromosomes*, que hace uso a su vez de una función nativa de LISP: *RASSOC***. Ésta recibe una A-lista y un valor y devuelve la clave asociada a dicho valor, con lo cual como mis claves son cromosomas y mis valores su puntuación al introducir (*RASSOC 0 *population**) obtengo automáticamente una solución óptima (la primera que este en la lista, siempre que exista). Explotando esto he creado la función *best_chromosomes*, que va incrementando el número de *fitness* y también comprueba si para un mismo número existe más de un cromosoma. El resultado es que **al ser *RASSOC* una función nativa de LISP, el tiempo de computo es infinitamente pequeño y la función *best_chromosomes* tarda muchísimo menos que *fitters_chromosomes***. Sumando esto a que sólo es llamada una vez por iteración y que sólo se realiza el calculo del *fitness*, en una iteración, sobre la mitad de la población obtenemos un algoritmo que es muy rápido, extremadamente eficiente y que explota al máximo lo que LISP puede ofrecer.

Resumiendo los cambios aportados:

- **Cambio en la estructura de la población**, ahora se almacenan los pares cromosoma y su puntuación *fitness* (generados aleatoriamente).

- **Tras llamar la primera vez a la función *exists_optimal_solution* se almacenan los dos mejores pares *cromosoma-fitness*, evitando tener que iterar más de una vez sobre la población.**
- **Se reduce a la mitad el número de veces que se llama a la función, que calcula el *fitness* de un cromosoma.**
- **Se usa la función nativa *RASSOC*, para explotar la nueva estructura de datos, obteniendo unos tiempos mínimos para conseguir los dos cromosomas mejores de la población.**

3.4.2 – Funciones implementadas

Las funciones escritas para llevar a cabo esta implementación son iguales que las anteriores y sólo cambian a nivel interno (ya que esta vez trabajan con pares de la A-lista, que es la población). Pero la estructura de las funciones es la misma que en las versiones anteriores, en parte porque creo, como dije en la introducción, que ésta es la gran potencia de los algoritmos genéticos; todos tienen la misma estructura aproximadamente, pero variando un poco por dentro las funciones se obtienen unos cambios notables. En esta sección habitualmente pongo el flujo de ejecución, asociando distintas funcionalidad a las funciones principales que he escrito, pero en este apartado repetiría por tercera vez el diagrama de flujo de las dos anteriores, sin aportar nada nuevo, ya que es el mismo. Para ver los cambios, hay que ver las funciones de los archivos “.lsp” donde estén meticulosamente explicadas.

3.4.3 – Información específica

La información relacionada con este algoritmo y sus restricciones siguen siendo las mismas que las del apartado anterior, sólo que esta vez la parte del algoritmo de enjambre (que ahora explicaré) si está implementada.

3.5 – Algoritmo de Enjambre

Este tipo de algoritmo es uno que nunca hemos visto en clase, se usa en este trabajo en caso de alcanzar un número de generaciones muy grande. Al ocurrir eso suponemos (y en el trabajo uso incorrectamente la afirmación), que el algoritmo genético no ha encontrado una solución, aunque quizás sí exista y necesite más tiempo para encontrarla.

Usar este algoritmo nos permite no devolver una solución exacta, sino mas bien una aproximada, que siempre es mejor que no devolver nada.

3.5.1 – Pseudocódigo

El pseudocódigo del algoritmo de enjambre es el que sigue:

```

expertChromosomes = best half of the final population;
aggregateChromosome = best performing chromosome;

for each (vertex in graph){
  if (vertex is part of a bad edge){
    newColor = most used color for vertex among
expertChromosomes;
    aggregateChromosomes.setColor(vertex, newColor);
  }
}

```

3.5.2 – Funciones Implementadas

Aunque haya tres implementaciones del algoritmo genético, de este algoritmo sólo he considerado hacer una, debido a que sólo en una de esas la eficiencia y los tiempos de ejecución eran buenos. Además porque, como ya explique antes, el gran problema computacional reside en la búsqueda y recorrido de la población y en este algoritmo también se lleva a cabo mucho, por lo que no sería nada eficiente haberlo escrito para los algoritmos genéticos más lentos, sólo para el más rápido. Además, el tiempo que tardan las dos primeras versiones del genético en llegar a lanzar este algoritmo es mucho.

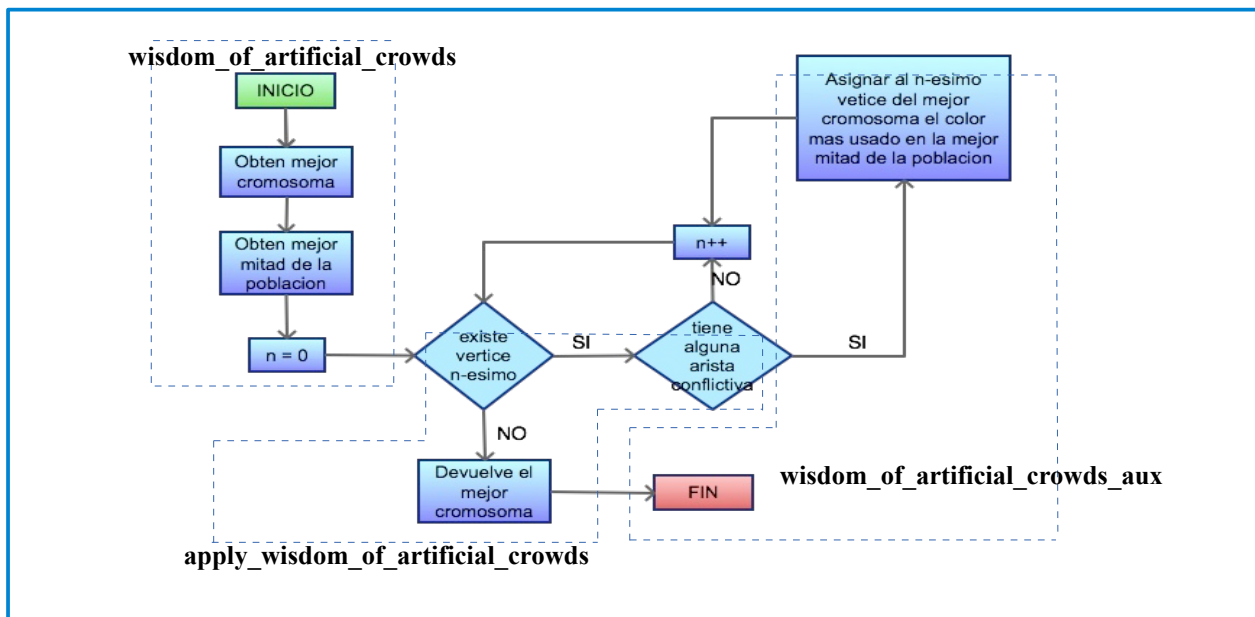
Por último y antes de comenzar a ver las funciones, me gustaría hablar de **la estructura de datos**. Como este algoritmo sólo está presente en el archivo del último genético “*hgenetic_algorithm_refined.lsp*” las estructuras de datos **sobre las que trabaja son las mismas pero, además, el algoritmo en sí requiere una serie de parámetros más:**

- ***expert_chromosomes*** : Contendrá la mitad de la población con mejor *fitness*.
- ***best_chromosome*** : Será el mejor cromosoma de la población.
- ***color_array*** : No es propiamente un parámetro del algoritmo, sino una variable útil a la hora de calcular el color más usado entre la mejor población media (*expert_chromosomes*).

Las funciones principales que he implementado han sido tres, para conocer más de ellas consultar el fichero correspondiente a este algoritmo. También hay implementadas una serie de funciones auxiliares, pero como ya están explicadas en el fichero, creo redundante volver a explicarlas aquí.

- *wisdom_of_artificial_crowds* : Es la función principal que inicia el algoritmo.
- *apply_wisdom_of_artificial_crowds*
- *wisdom_of_artificial_crowds_aux*

En lugar de exponerlas una a una, considero que la mejor forma de ver y resumir lo que hacen es mostrando el diagrama de flujo del algoritmo en sí y asignándole a cada función una porción de funcionamiento:



Como ya dije antes, se puede ver que se itera mucho sobre la población, aunque luego, a la hora de los resultados gracias a la estructura de datos usada realmente los tiempos son muy lentos.

3.5.3 – Información específica.

Este algoritmo en principio no tiene detalles intrínsecos a su implementación, principalmente porque juega un papel auxiliar y, por lo tanto, sufre las mismas restricciones que la tercera versión del algoritmo genético.

4.0 – Algoritmos Genéticos Paralelos

La idea de los algoritmos genéticos paralelos es **explotar la capacidad multiprocesador de las arquitecturas**. Con ese fin se han desarrollado una serie de modelos para poder “subdividir” los problemas y poder repartirlos entre distintos hilos. Cabe destacar que, **aunque no se ejecuten varios hilos y solo se subdivide un problema, ya se obtienen unos tiempos muy buenos, puesto que se estaría aplicando una especie de “divide y vencerás”**. Debido a la complejidad que supone programar hilos, en el presente trabajo he enfocado este punto más por la rama de “divide y vencerás”, que la explotación multiprocesador, que además requiere conocer la versión específica del interprete que se está usando (la cual será, en el momento de la corrección, desconozco). **Por todo esto, no he escrito un código paralelo, pero si fácilmente paralelizable**.

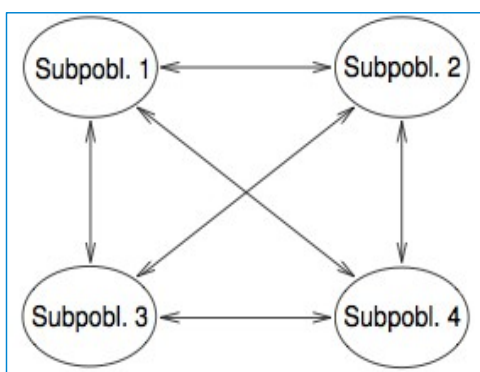
4.1 – ¿Como funciona el modelo de islas?

El **concepto clave** del modelo de islas es **dividir la población total en pequeñas subpoblaciones, que serán las islas, y en cada una de ellas llevar a cabo un algoritmo genético**. Cada cierto número de generaciones se produce un fenómeno que se denomina **emigración**. **Consiste en escoger una serie de individuos de cada isla y trasladarlos a otra**. Este fenómeno **permite explotar las diferencias entre las distintas poblaciones de las islas, obteniendo una fuente de diversidad genética muy grande**. En base a la comunicación entre las islas podemos definir distintos tipos de modelo de islas:

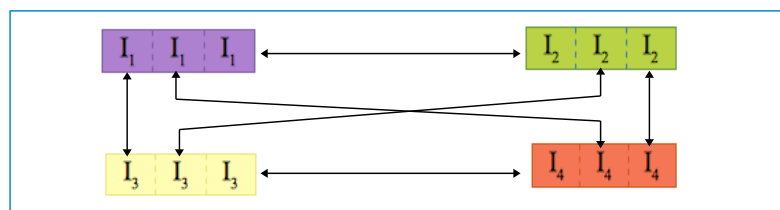
- Comunicación en estrella
- Comunicación en anillo
- Comunicación en red.

En el trabajo he usado esta última, por lo que me centraré en explicarla. Para tener información sobre las otras consultar el documento relacionado con este tema, presente en la bibliografía y la webgrafía.

En la comunicación en anillo no existe una jerarquía entre las subpoblaciones. Cada una mandará una serie de hijos a las demás, siguiendo el siguiente esquema:



En el trabajo, el número de individuos que se envían son tres por isla, de tal forma que se envía un cromosoma a cada vecino. El esquema de emigración es el siguiente:



Añadir este modelo al trabajo me pareció buena idea, porque en clase no hemos visto nada parecido y realmente me parece una herramienta potente y eficaz, pero sobre todo muy simple de implementar.

4.2 – Representación del problema.

Tras programar los algoritmos de la primera parte, me quedo claro que la mejor estructura de datos para operar fue la usada en la tercera implementación. Realmente **en esta sección no se introduce ningún elemento nuevo, simplemente se divide una población. Dicha población será una lista de cromosomas normal y corriente, pero se verá modificada para adaptarse a la estructura de datos de la tercera versión del algoritmo genético híbrido**, recordemos los elementos que teníamos en un problema de ese tipo:

- **Cromosomas** : Siguen siendo una lista de números.
- **Colores** : Siguen siendo números.
- **Vértices** : Sigue siendo un número.
- **Generación**: Sigue siendo un número.
- **Máxima generación**
- **Grafo**: Sigue usándose la matriz de adyacencia.
- **Población**: Se usa una lista de pares *cromosoma-puntuacion* o una A-Lista en cada isla, pero el algoritmo recibe una población, que es una lista de cromosomas a secas.

Como vemos, por lo tanto, no se usa ninguna estructura de datos nueva. Simplemente una distribución distinta. **El algoritmo del modelo de islas recibirá una lista de cromosomas normales y las dividirá en subpoblaciones (islas), que seguirán el esquema de datos de la población descrita en los puntos (A-Lista).**

4.3 – Pseudocódigo

Como en este apartado tenemos el modelo de islas y el algoritmo genético correspondiente hablaremos de ambos empezando por el modelo de islas, su pseudocódigo es el que sigue:

```
funcion modelo_islas ( poblacion){
    islas[] = divide_en_islas(poblacion);
    solucion = null;
    generacion = 0;
    While( generacion < max_num_generaciones ){
        if ( generacion de emigracion == generacion)
            emigrar();
        algoritmo_genetico( isla[0]);
        algoritmo_genetico( isla[1]);
        algoritmo_genetico( isla[2]);
        algoritmo_genetico( isla[3]);
        if (existe_solucion_en_islas ( islas) )
            solucion = get_solucion (islas);
        generacion++;
    }
    return solucion;
}
```


Como vemos, realmente el modelo de islas sólo aporta la división de la población que recibe como parámetro de entrada y la emigración. Con ésto último obtenemos una gran diversidad genética, es por eso que en esta sección realmente he implementado dos algoritmo genéticos distintos. Son prácticamente iguales, pero varían a la hora de crear una nueva generación. El motivo de esto es que me pareció que se aportaría una gran diversidad genética, de esta manera y gracias a la emigración todas las islas gozarían de los beneficios de esto.

Los algoritmo genéticos son así:

```
function genetico(poblacion){
    padres[] = seleccionaPadres;
    hijo = cruza(padres[0], padres[1]);
    hijo = muta( hijo);
    añadeloALaPoblacion(hijo);
    return poblacion;
}
```

- La principal diferencia entre los dos algoritmos radica en la función que equivaldría a “añadeloALaPoblacion”. El primero lo que hará será sustituir el peor individuo de la población por el hijo, este algoritmo es el que usa la isla uno y la isla dos. El otro lo que hará será coger la primera mitad mejor de la población, sustituir de ella el peor individuo por el hijo y rellenar el resto con cromosomas aleatorios.

Implementando los dos genéticos de esta manera, mi intención era explotar la propiedad de la emigración, ya que las islas uno y dos se centrarán en poseer una población más elitista, mientras que, gracias al rellenado aleatorio, las islas tres y cuatro poseerán una población muy variada. Y gracias a la emigración, todas las islas podrán compartir sus ventajas.

El pseudocódigo de cada función se corresponde con los del algoritmo genético de la sección anterior de la siguiente manera:

- **seleccionaPadres** es la misma función que *parentselection2* .
- **cruza** equivale a *crossover*.
- **muta** es igual que *mutation1*.

```
function añadeloALaPoblacion_1 (hijo){
    poblacion.OrdenaAscendentemente();
    poblacion.EliminaUltimoElmento();
    poblacion.add(hijo);
}
```

```
function añadeloALaPoblacion_2 (hijo){
    poblacion.OrdenaAscendentemente();
    poblacion.BorraMitadInferior();
    poblacion.EliminaUltimoElmento();
    poblacion.add(hijo);
    poblacion.rellenaCnCromosomasRandom()
}
```

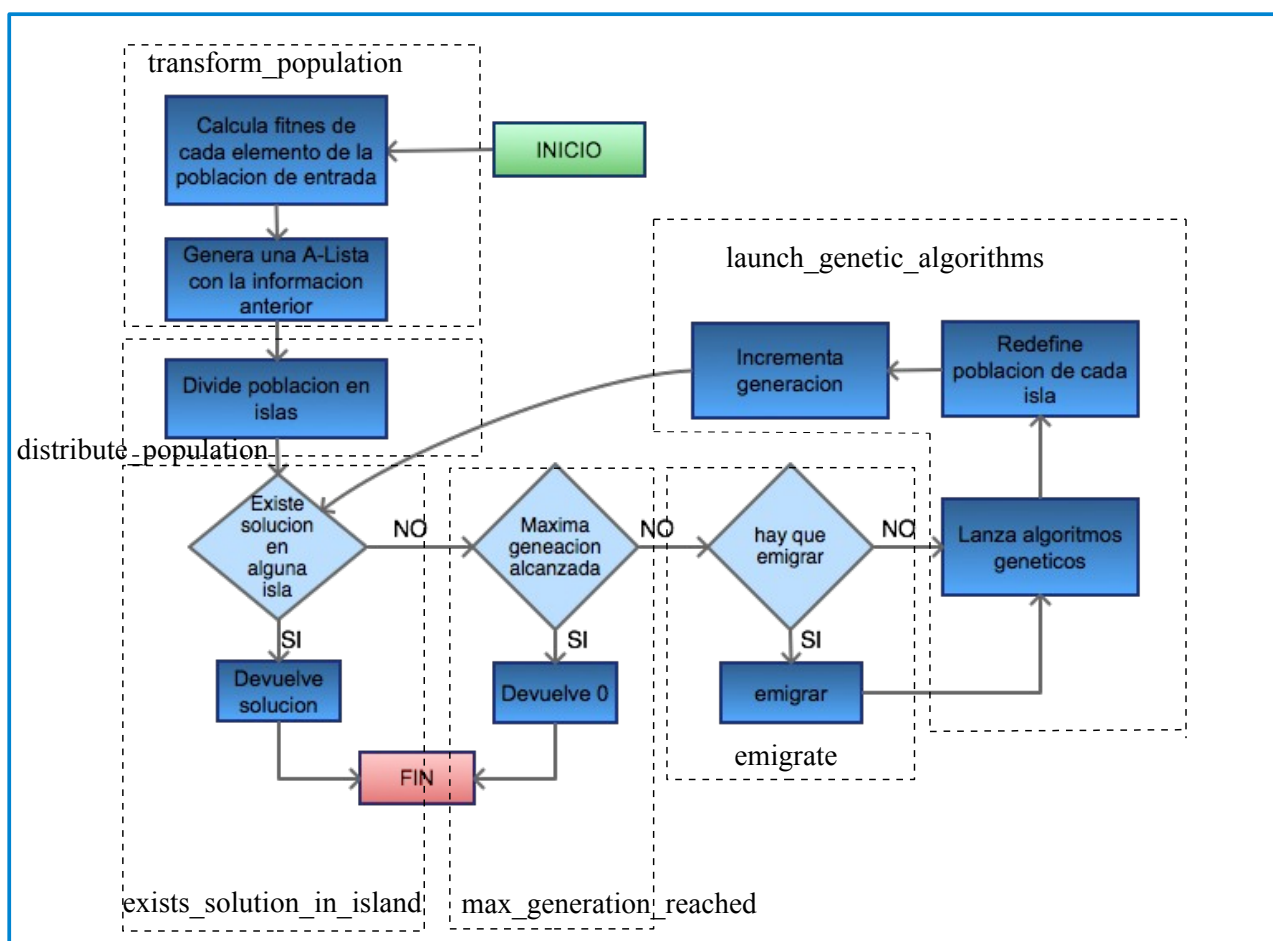
Como vemos en el pseudocódigo aunque yo no haya implementado un paralelismo real, sino sólo la división del problema, lo he planteado de tal forma que, en un momento dado, lanzar cada algoritmo genético en un hilo se volvería muy fácil. Cada uno se asigna a un procesador y se ejecutan independientemente sobre la población de cada isla. Si se alcanza una generación de emigración se espera a que todos los hilos terminen, se actualiza cada isla y se vuelve a empezar.

4.4 – Funciones implementadas

Las funciones principales que he escrito para llevar a cabo la funcionalidad del modelo de islas han sido⁴:

- *distribute_population*
- *transform_population*
- *emigrate*
- *exists_solution_in_island*
- *max_generation_reached*
- *start_island_model*: Engloba las dos primeras funciones, lanza e algoritmo.
- *start*: engloba las funciones de control de soluciones, emigración y generaciones.
- *lauch_genetics_algorithms*: se encarga de lanzar cada algoritmo genético y redefine la población de cada isla con la salida de cada algoritmo, aquí es donde se paralelizaría el código.

La mejor forma de ver lo que realiza cada una es con el flujo de ejecución:

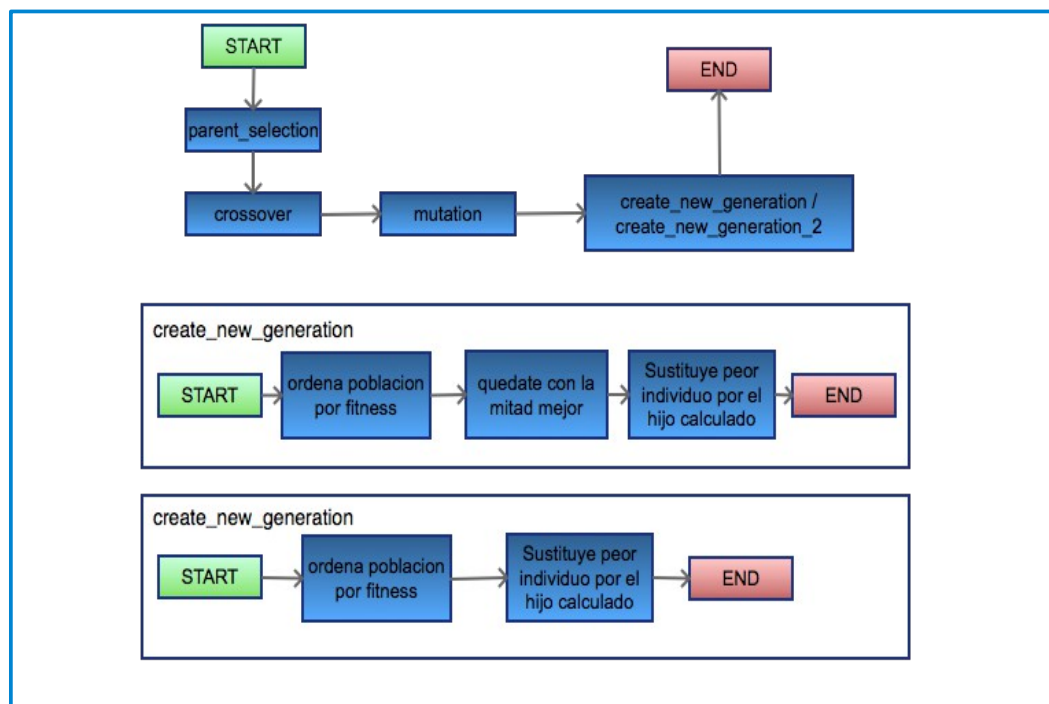


⁴ Para leer la descripción de cada una consultar el fichero "islands_model.lsp"

Por otro lado para el algoritmo genético:

- *parent_selection*
- *crossover*
- *mutation*
- *create_new_generation* : Correspondería a la que coge la mitad mejor de la población, sustituye el peor individuo por el hijo y rellena el resto aleatoriamente.
- *create_new_generation_2* : Sustituye solo el peor individuo con el hijo.
- *genetic_algorithm*: lanza las funciones anteriores con la función *create_new_generation*.
- *genetic_algorithm_2* : Idem con la segunda función de generación de generaciones.

Una vez más, ilustraré las funciones en un diagrama de flujo, sólo que esta vez en casi todos los recuadros estarán rellenos con el nombre de la función correspondiente porque la ejecución es, como se verá, muy lineal:



4.5 – Información específica.

En la implementación realizada, hay una serie de restricciones tácitas que es necesario conocer y especificar:

- El algoritmo recibe como entrada una lista de cromosomas.
- La población mínima es de 12, ya que se necesitan como mínimo tres cromosomas para cada isla. Pero para poblaciones mayores el algoritmo no necesita que se introduzcan múltiplos de 4 para funcionar, ya que en caso de que la población no sea

divisible entre las islas rellena lo que falta con cromosomas generados aleatoriamente, de tal forma que evita que el usuario tenga que preocuparse de estar controlando el tamaño de población que introduce.

- En caso de no encontrar una solución devolverá un 0.
- En caso de que se produzca un error durante la ejecución el algoritmo devolverá NIL.
- En los ejemplos del apartado *Benchmarks* la población que recibe de entrada el algoritmo es generada de forma aleatoria, aunque podría introducirse una que no lo fuera.

5.0 – Benckmarks.

En este apartado voy a exponer distintos datos. Para empezar cada subapartado pertenece a un tipo de grafo. En cada uno expondré un dibujo del mismo, el contenido del fichero “.txt” asociado, el mínimos de colores necesarios para su resolución, la matriz de adyacencia obtenida con el *script* y necesaria para resolver la coloración y, por último, una tabla de tiempos de computación variando los parámetros. En la tabla aparecerá sólo un tiempo de ejecución, pero en realidad es la media de 5 ejecuciones, ya que como los algoritmos tienen una cierta componente aleatoria. Creo que esta es la mejor manera de conocer la eficiencia de los mismos.

Una cosa que pretendo ilustrar en este apartado es la eficiencia de cada algoritmo previamente descrito, para ello **los problemas van en creciente dificultad de resolución**, ya que en un problema pequeño a lo mejor las diferencias serán mínimas entre los resultados, pero conforme se aumente se irán acentuando más. Esto será otro factor que denotará más aún esa evolución de la que hablaba en la introducción.

Por último, el equipo usado para las pruebas es un Mac OS i7 2.8 GHZ con 16 GB de RAM DDR3 a 1333 MHZ.

5.1 – Grafos usados en el trabajo

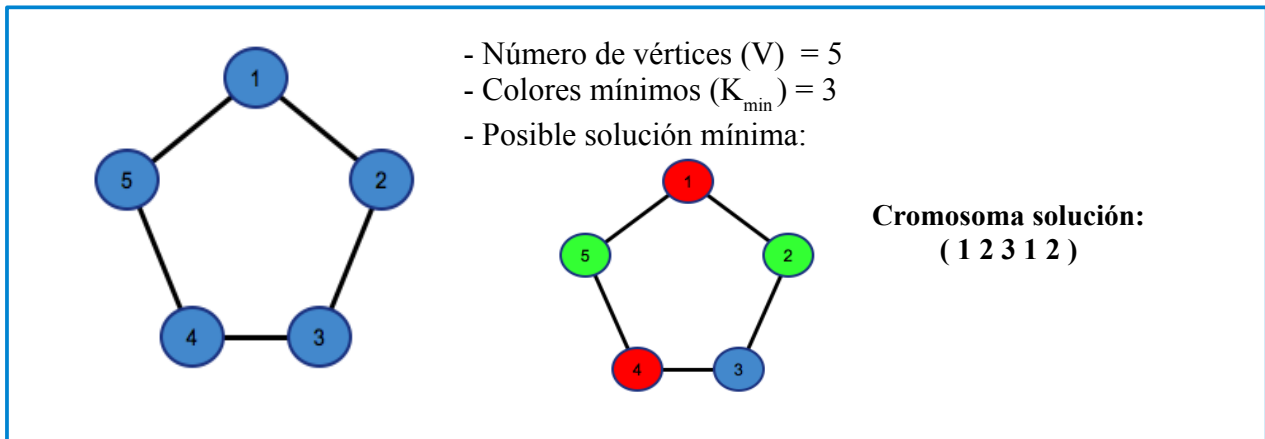
Los grafos que usaré y expondré son los siguientes:

- **Grafo Pentágono** : Es un ciclo 5, lo he añadido como el problema más simple para resolver.
- **Grafo de los puentes de Königsberg**.
- **Grafo de Petersen** : Es un famoso grafo, denominado grafo de Petersen⁵.
- **Grafo del mapa de Andalucía** : Es el mapa de Andalucía transformado en grafo.
- **Grafo Bipartito**.

5 http://es.wikipedia.org/wiki/Grafo_de_Petersen

5.2 – Grafo Pentágono.

El grafo de este apartado sería el contenido en el fichero “*graph_pentagon.txt*” :



Una vez vista la forma que tiene, veamos el fichero que lo contiene y su transformación al archivo “*data.lsp*” (matriz de adyacencia), usando los colores mínimos.

```
c FILE: graph_pentacle.txt
c
c Created by: Andrea Cimmino Arriaga
c
c This is a five transformed in a graph.
p edge 5 10
e 1 4
e 1 3
e 2 5
e 2 4
e 3 1
e 3 5
e 4 1
e 4 2
e 5 2
e 5 3
```

```
(defparameter *v* 5)
(defparameter *k* 3)
(defparameter *matrix_adj* '( ('(0 0 1 1 0)
                                '(0 0 0 1 1)
                                '(1 0 0 0 1)
                                '(1 1 0 0 0)
                                '(0 1 1 0 0)
                                ) )
```

De esta manera, el problema de colorear un grafo ciclo 5 (un pentágono) ya está formalizado y sólo falta lanzar los algoritmos y ver sus resultados. Esta última parte corresponde a la siguiente tabla:

- K es el número de colores usado.
- L es el tamaño de la población.
- GM es el número de generaciones máximas que iterará el algoritmo.
- El parámetro de emigración en el primer modelo de islas es de 4 generaciones y en el segundo es de 8.

Algoritmos Parámetros	Genético híbrido versión1	Genético híbrido versión2	Genético híbrido versión3	Modelo de islas	Modelo de islas
[K = 3, L =50, GM= 20.000]	0.00068 seg	0.00057 seg	0.00001 seg	0.000015 seg	0.000023 seg
[K = 3, L =2, GM= 20.000] ⁶	0.000385 seg	0.000233 seg	0.000051 seg	0.000042 seg	0.000055 seg
[K = 2, L =50, GM= 20.000]	17.247235 seg	8.771565 seg	1.299869 seg	1.092737 seg	0.911154 seg

Por último, me gustaría comentar un poco los resultados. Para realizar cada prueba se ha usado los mismos parámetros para intentar poner en igual dificultad a los algoritmos, todas las poblaciones iniciales han sido generadas aleatoriamente.

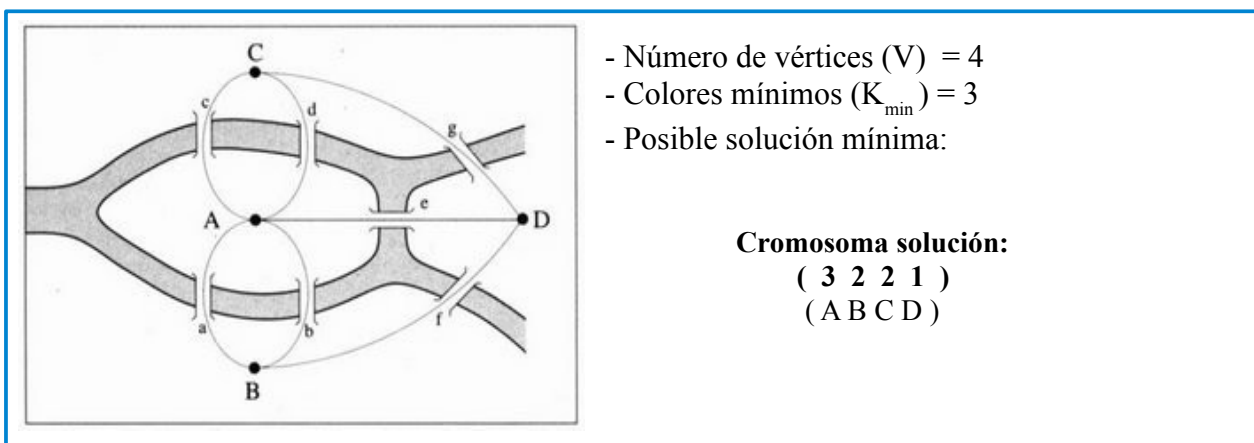
Cada prueba ha sido puesta por un motivo. **La primera es para ver el algoritmo en condiciones normales**, vemos que todos obtienen unos tiempos bajísimos, aunque ya se notan las diferencias. Esto se debe a que la población es muy grande y la generación aleatoria ya aporta la solución, por lo general solo hay que buscarla. Debido a esto en **la segunda prueba la población se reduce al mínimo para que el algoritmo tenga que iterar, de esta manera se ve mejor el tiempo de iteración media de cada algoritmo**. Vemos que los tiempos siguen siendo bajos, pero ya se van diferenciando. **El tercer ejemplo está puesto para ver, como mucho, cuánto tardaría cada algoritmo, si no encuentra solución**. En el caso de la versión 3 del algoritmo híbrido se obtiene una aproximación de la solución, en este ejemplo se ve claramente la velocidad de cada algoritmo y la diferencia de uno a otro.

Se aprecia que los mejores tiempos son los obtenidos con el modelo de islas y con la tercera implementación del algoritmo híbrido, esto era de esperar. Aún así, para este poco volumen de datos no se aprecia gran diferencia entre los dos y, en general, los tiempos son más bajos. En el siguiente ejemplo aumentaremos un poco el número de nodos y de aristas para ver cómo se comportará cada algoritmo

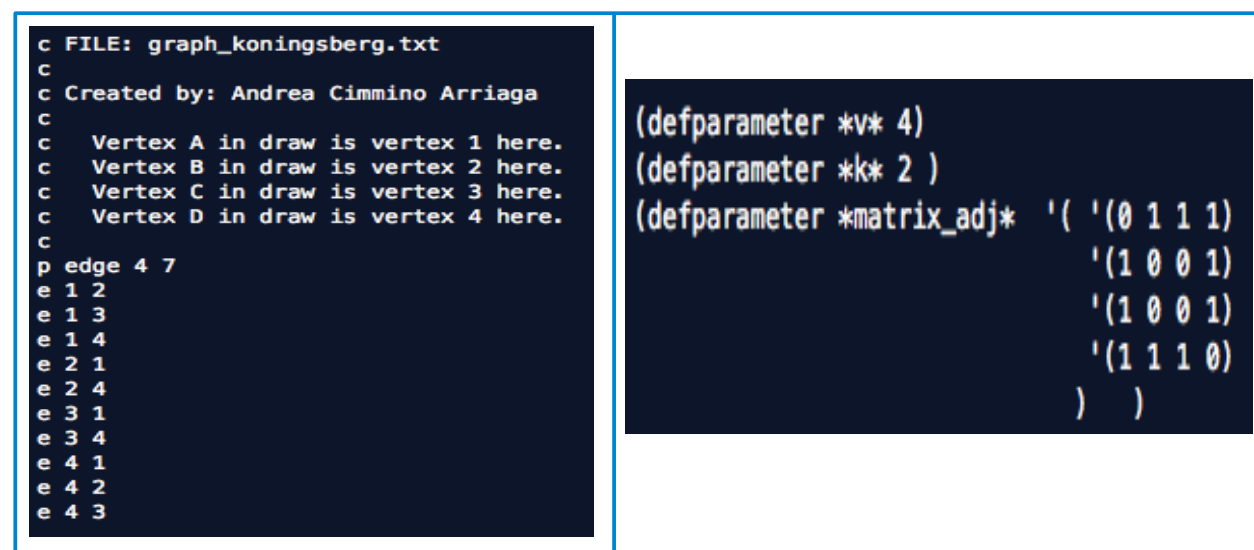
6 En este caso para las islas la población mínima fue de 12.

5.3 Grafo de los puentes de Konigsberg

Este grafo está contenido en el fichero “*graph_konigsberg.txt*”. Es un famoso grafo que representa el problema de los siete puentes de Konigsberg⁷. Su representación sería:



A continuación veamos el fichero que lo contiene y su transformación al archivo “*data.lsp*” (matriz de adyacencia), usando los colores mínimos:



Con los datos anteriores ya tenemos formalizado el problema y solo falta ejecutar cada algoritmo y ver los resultados que obtienen:

- K es el número de colores usado.
- L es el tamaño de la población.
- GM es el número de generaciones máximas que iterará el algoritmo.
- El parámetro de emigración en el primer modelo de islas, es de 4 generaciones y en el segundo es de 8.

⁷ http://es.wikipedia.org/wiki/Problema_de_los_puentes_de_K%C3%B6nigsberg

Algoritmos Parámetros	Genético híbrido versión1	Genético híbrido versión2	Genético híbrido versión3	Modelo de islas	Modelo de islas
[K = 3, L =50, GM= 20.000]	0.000801 seg	0.001145seg	0.000023 seg	0.000164 seg	0.000219 seg
[K = 3, L =2, GM= 20.000] ⁸	0.001726 seg	0.00112 seg	0.000055 seg	0.000041 seg	0.000046 seg
[K = 2, L =50, GM= 20.000]	13.607057 seg	6.391777 seg	1.089592 seg	1.141901 seg	0.769600 seg

Como anteriormente, vemos que los tiempos son relativamente bajos, donde más se aprecia la velocidad de cada algoritmo es en la tercera prueba, en la que vemos claramente cuánto tarda un algoritmo en procesar todos los datos las 20.000 generaciones. Se puede apreciar cómo los tiempos en las implementaciones de los algoritmos genéticos híbridos se van reduciendo muy drásticamente. Eso se debe a todas las mejoras que se le han ido añadiendo a las implementaciones sucesivas.

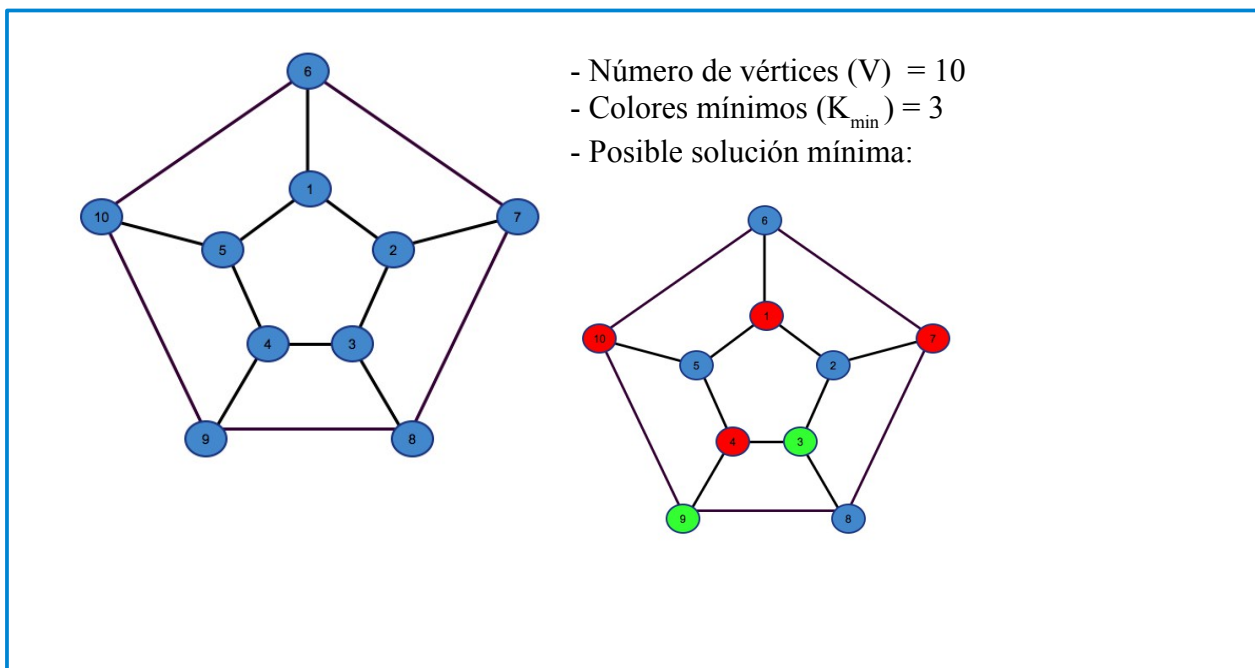
Por otro lado, en el modelo de islas vemos que la tasa de emigración no afecta tanto a los tiempos, quizás si la tasa es más grande se procesa más rápido todo (en el caso de no encontrar solución), lo cual es lógico porque se llama menos veces a la función de migración, por otro lado, vemos que emigrar hace que la solución se encuentre en relativamente menos tiempo.

Comparando la tercera versión del algoritmo genético híbrido y el modelo de islas vemos que la diferencias no son muy grandes. La principal es a la hora de cambiar la población, cuanto mayor es como el modelo tiene que adaptar la población aleatoria entrante a las islas y tiene que hacer un procesado tarda un tiempo que se ve claramente reflejado en la tabla. Aún así dicho tiempo es muy aceptable, ya que la diferencia no es tan grande y esto se ve en la tercera prueba donde tienen que procesar todos los datos, en ese caso la aleatoriedad de los algoritmo influye menos, ya que nunca se alcanza una solución gracias a ella y vemos que los algoritmos genéticos si se diferencian entre si, mientras que el modelo de islas y la tercera implementación de los anteriores no se diferencian.

8 En este caso para las islas la población mínima fue de 12.

5.4 – Grafo de Petersen.

Este grafo es el contenido en el archivo “*graph_petersen.txt*” y es así:



Igual que antes, veamos el archivo que contiene al grafo y su matriz de adyacencia:

```
c FILE: graph_petersen.txt
c
c SOURCE: Andrea Cimmino Arriaga
c DESCRIPTION: the famous graph of petersen
c
p edge 10 15
e 1 4
e 1 3
e 1 6
e 2 5
e 2 4
e 2 7
e 3 1
e 3 5
e 3 8
e 4 1
e 4 2
e 4 9
e 5 2
e 5 3
e 5 10
e 6 1
e 6 7
e 6 10
e 7 2
e 7 6
e 7 8
e 8 3
e 8 7
e 8 9
e 9 4
e 9 8
e 9 10
e 10 5
e 10 6
e 10 9
```

```
(defparameter *v* 10)
(defparameter *k* 3)
(defparameter *matrix_adj* '( (0 0 1 1 0 1 0 0 0 0)
                               (0 0 0 1 1 0 1 0 0 0)
                               (1 0 0 0 1 0 0 1 0 0)
                               (1 1 0 0 0 0 0 0 1 0)
                               (0 1 1 0 0 0 0 0 0 1)
                               (1 0 0 0 0 0 1 0 0 1)
                               (0 1 0 0 0 1 0 1 0 0)
                               (0 0 1 0 0 0 1 0 1 0)
                               (0 0 0 1 0 0 0 1 0 1)
                               (0 0 0 0 1 1 0 0 1 0)
                               ) )
```

Con los datos anteriores ya tenemos formalizado nuestro problema, pasemos a ver la tabla de resultados obtenida:

- K es el número de colores usado.
- L es el tamaño de la población.
- GM es el número de generaciones máximas que iterará el algoritmo.
- El parámetro de emigración en el primer modelo de islas es de 4 generaciones y en el segundo es de 8.

Algoritmos Parámetros	Genético híbrido versión1	Genético híbrido versión2	Genético híbrido versión3	Modelo de islas	Modelo de islas
[K = 3, L =50, GM= 20.000]	0.002402 seg	0.003261 seg	0.000486 seg	0.000533 seg	0.000655 seg
[K = 3, L =2, GM= 20.000] ⁹	0.000777 seg	0.0004 seg	0.000414 seg	0.000315 seg	0.000282 seg
[K = 2, L =50, GM= 20.000]	47.676418 seg	26.025823 seg	3.362560 seg	2.516653 seg	2.035825 seg

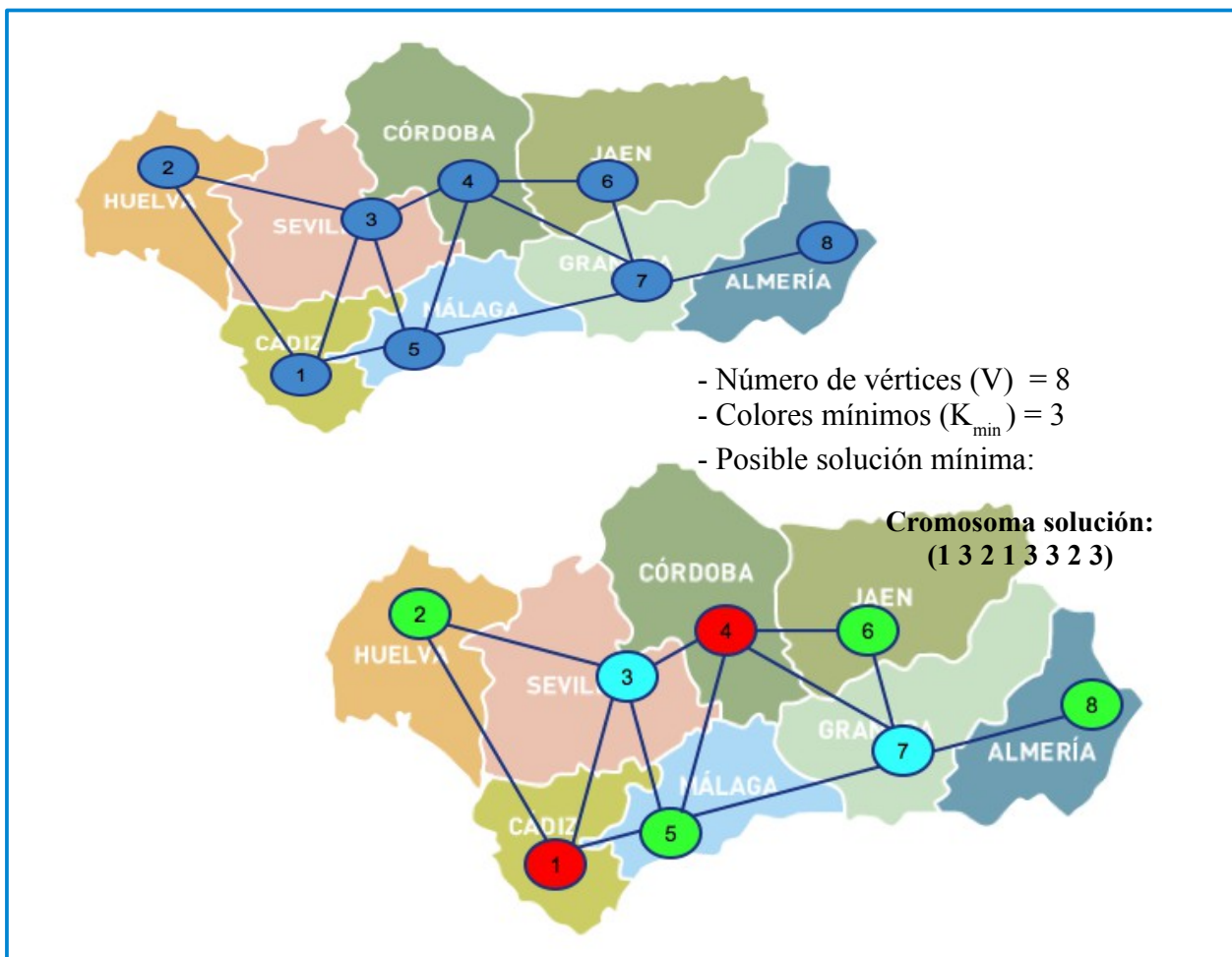
Vemos que las pruebas son con la misma variación de parámetros que antes.

En primer lugar vemos que los algoritmo genéticos híbridos empiezan, en todas las pruebas a distanciarse entre ellos con una diferencia sustancial. Por otro lado la versión definitiva y el modelo de islas poseen unos tiempos relativamente parecidos, una cosa que llama la atención al ver estos resultados es que al reducir la población al mínimo los algoritmos genéticos de la versión uno y dos, tardan menos que si usaran una población mayor, esto tiene su lógica ya que el cuello de botella de los algoritmos es recorrer la población, al ser esta menor tardan mucho menos. Por otro lado este fenómeno también se aprecia en el resto, pero las diferencias no son tan grandes. Aunque hay que remarcar que al haber mucha componente aleatoria también se han dado caso en las pruebas que en la generación aleatoria inicial de la población ya existiera una solución. Así que los tiempos mínimos son muy relativos a la “suerte” que se tenga con dicha generación. Pero donde realmente se ve la potencia de cada algoritmo es en la tercera prueba, en la cual la aleatoriedad no influye (ya que no existe solución) por lo que ahí es donde realmente se ve el tiempo máximo que tardaría un algoritmo de los anteriores. Una cosa que también me gustaría comentar son los dos modelos de islas, ya que pensé que los tiempos se diferenciarían mucho si cambiaba el parámetro de emigración, pero se puede apreciar que realmente cambian muy muy poco en este caso, aunque como veremos más adelante puede llegar a ser sustancial.

9 En este caso para las islas la población mínima fue de 12.

5.5 – Grafo del mapa de Andalucía.

Este grafo esta contenido en el fichero “*graph_andalucia.txt*” y es como sigue:



Igual que anteriormente, veamos la formalización del problema con los archivos relacionados:

```
c
c SOURCE: Andrea Cimmino Arriaga
c DESCRIPTION: It's the andaluci's map translated to a graph.
c Nodes description:
c   - CADIZ: 1
c   - HUELVA: 2
c   - SEVILLA: 3
c   - CORDOBA: 4
c   - MALAGA: 5
c   - JAEN: 6
c   - GRANADA: 7
c   - ALMERIA: 8
c
p edge 8 246708
e 1 2
e 1 3
e 1 5
e 2 1
e 2 3
e 3 1
e 3 2
e 3 4
e 3 5
e 4 3
e 4 5
e 4 6
e 4 7
e 5 1
e 5 3
e 5 4
e 5 7
e 6 4
e 6 7
e 7 4
e 7 5
e 7 6
e 7 8
e 8 7
```

```
(defparameter *v* 8)
(defparameter *k* 3)
(defparameter *matrix_adj* '( (0 1 1 0 1 0 0 0)
                               (1 0 1 0 0 0 0 0)
                               (1 1 0 1 1 0 0 0)
                               (0 0 1 0 1 1 1 0)
                               (1 0 1 1 0 0 1 0)
                               (0 0 0 1 0 0 1 0)
                               (0 0 0 1 1 1 0 1)
                               (0 0 0 0 0 0 1 0)
                               ) )
```

Con los datos anteriores ya tenemos formalizado nuestro problema. Pasemos a ver la tabla de resultados obtenida:

- K es el número de colores usado.
- L es el tamaño de la población.
- GM es el número de generaciones máximas que iterará el algoritmo.
- El parámetro de emigración en el primer modelo de islas es de 4 generaciones y en el segundo es de 8.

Algoritmos Parámetros	Genético híbrido versión1	Genético híbrido versión2	Genético híbrido versión3	Modelo de islas	Modelo de islas
[K = 3, L =50, GM= 20.000]	0.030984 seg	0.017014 seg	0.003791 seg	0.000520 seg	0.000593 seg
[K = 3, L =2, GM= 20.000] ¹⁰	0.00720 seg	0.00382 seg	0.000337 seg	0.000210 seg	0.000263 seg
[K = 2, L =50, GM= 20.000]	36.522827 seg	17.910454 seg	2.579846 seg	2.078656 seg	1.780560 seg

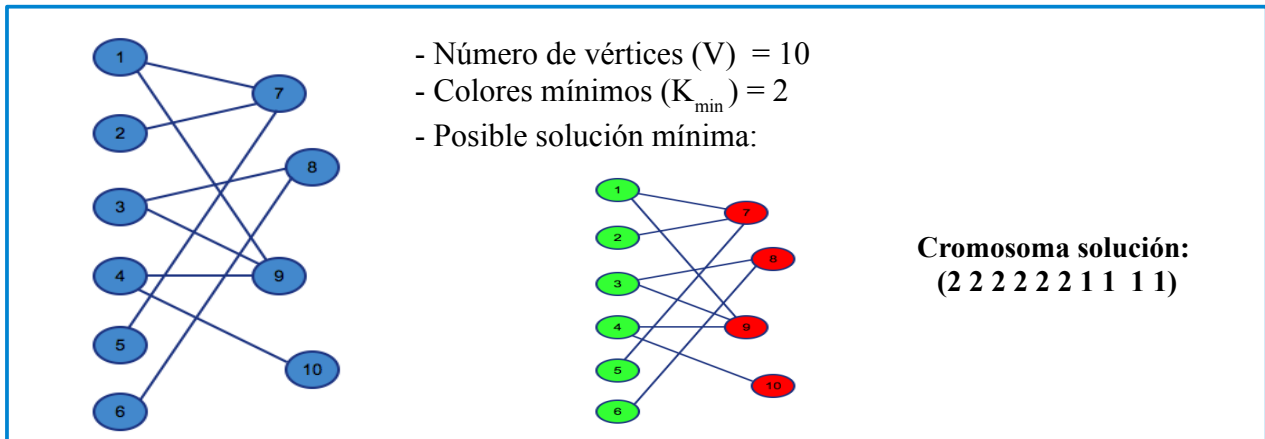
Lo primero que hay que comentar de estos tiempos es que en la segunda prueba, al reducir la población, el algoritmo genético híbrido versión 3 no suele converger a la solución, de las cinco ejecuciones sólo convergió una, de ahí el tiempo tan alto.

Por otro lado, sorprende ver que en los modelos de islas, para una población mínima la velocidad es mayor. La explicación de esto es la siguiente: para empezar ambos algoritmos convergen mucho más rápido que el resto, por lo general llegan a encontrar la solución con una o dos generaciones como mucho (en este problema en concreto), por lo que el tiempo es debido a tener que trabajar con mayor número de individuos (también influye, como digo en cada apartado, la aleatoriedad de los algoritmos). Por otro lado, se aprecia que reduciendo los tiempos de emigración el tiempo máximo de de ejecución baja bastante.

¹⁰ En este caso para las islas la población mínima fue de 12.

5.6 - Grafo Bipartito

El grafo de este apartado corresponde al del fichero “*graph_bipartite.txt*”, y es así:



Veamos ahora sus ficheros asociados:

```
c FILE: graph_koningsberg.txt
c
c Created by: Andrea Cimmino Arriaga
c
p edge 10 9
e 1 7
e 1 9
e 2 7
e 3 8
e 3 9
e 4 9
e 4 10
e 5 7
e 6 8
e 7 1
e 7 2
e 7 5
e 8 3
e 8 6
e 9 1
e 9 3
e 9 4
e 10 4

(defparameter *v* 10)
(defparameter *k* 2 )
(defparameter *matrix_adj* '( (0 0 0 0 0 0 1 0 1 0)
                                (0 0 0 0 0 0 1 0 0 0)
                                (0 0 0 0 0 0 0 1 1 0)
                                (0 0 0 0 0 0 0 0 1 1)
                                (0 0 0 0 0 0 1 0 0 0)
                                (0 0 0 0 0 0 0 1 0 0)
                                (1 1 0 0 1 0 0 0 0 0)
                                (0 0 1 0 0 1 0 0 0 0)
                                (1 0 1 1 0 0 0 0 0 0)
                                (0 0 0 1 0 0 0 0 0 0)
                                ) )
```

Una vez formalizado el problema, e igual que en los apartados anteriores, veamos los resultados obtenidos expuestos en la siguiente tabla:

- K es el número de colores usado.
- L es el tamaño de la población.
- GM es el número de generaciones máximas que iterará el algoritmo.
- El parámetro de emigración en el primer modelo de islas es de 4 generaciones y en el segundo es de 8.

Algoritmos Parámetros	Genético híbrido versión1	Genético híbrido versión2	Genético híbrido versión3	Modelo de islas	Modelo de islas
[K = 2, L =50, GM= 20.000]	0.157467 seg	0.037634 seg	0.000439 seg	0.000839 seg	0.000692 seg
[K = 2, L =2, GM= 20.000] ¹¹	0.00658 seg	0.00257 seg	0.000457seg	0.000667 seg	0.000517 seg
[K = 1, L =50, GM= 20.000]	49.613857 seg	25.399936 seg	3.759576 seg	3.597117 seg	3.722441 seg

En este problema lo primero que salta a la vista es que la primera implementación del algoritmo híbrido encuentra la solución antes, con una población menos que con una mayor, que es lo contrario que cabría esperar. Esto se debe a dos cosas, la primera es que la componente aleatoria de estos algoritmos, es este caso en las 5 ejecuciones el algoritmo encontró la solución en breves iteraciones, mientras en las 5 con una amplia población tardo mucho más. La segunda es la propia implementación, recordemos que el cuello de botella de estos algoritmos era iterar sobre la población, así que a menor población menor tiempo que tarda en realizar una iteración.

Luego, como en los ejemplos anteriores, vemos que la tercera implementación tiene unos tiempos mejores que las versiones anteriores, pero que se parecen mucho a los tiempo obtenidos con el modelo de islas. Más o menos vemos que este es un patrón que se ha ido repitiendo en todos los ejemplos.

5.7 - Problemas de optimización que no son de coloración.

En este apartado debería de presentar una serie de problemas de optimización que se resuelvan con los algoritmos implementados y que sean distintos a la coloración. Por ejemplo el problema de las n-reinas o encontrar el mínimo de la función cuadrado.

Por desgracia, cuando comencé este trabajo no tuve en cuenta este punto y elegí desarrollar un algoritmo (el genético híbrido) que estaba creado específicamente para el coloreado de grafos. Y como el siguiente algoritmo (modelo de islas) está basado en este primero, me fue imposible realizar este apartado, es decir, podría adaptar el algoritmo de Musa M. Hindi y Roman V. Yampolskiy para que resuelvan problemas genéticos, pero no sería el mismo algoritmo, ya que tiene puntos (como la toma de decisión de qué padres seleccionar o que mutación realizar, que dependen exclusivamente de los colores y no tendría sentido en otros problemas).

¹¹ En este caso para las islas la población mínima fue de 12.

6.0 - Conclusiones

A la vista de los resultados obtenidos en los apartados anteriores, vemos que **la implementación tercera del algoritmo híbrido supera bastante a las dos anteriores**, y eso que los problemas que he expuesto son pequeños. Por otro lado, probé a lanzar los algoritmos sobre el archivo `flat1000_50_0.col`¹² de la web de *benchmarks*¹³ indicada en la web de la asignatura. Los resultados de esa ejecución no los he incluido, porque la tercera versión de este algoritmo tras media hora de trabajo todavía estaba en la generación número 476 y no encontraba una solución. Esto se debe a que el volumen de datos de ese fichero es muy muy grande (1000 vértices). Pero digo esto porque, **aunque los tiempos expuestos antes son muy pequeños, quiero que quede claro que es porque están contextualizados a problemas mucho más pequeños y que en el tiempo de ejecución influye mucho el volumen de datos que ha de manejarse por iteración**. Es precisamente en ese tipo de problemas donde más se ve la diferencia entre las implementaciones, ya que las dos primeras solo para realizar una, tardaban con cinco veces más que la tercera.

Por otro lado en el modelo de islas, personalmente, me surgió la duda de si era mejor poner una tasa de emigración cada pocas generaciones o, por el contrario, cada muchas. Desgraciadamente la aleatoriedad puede llevar a error a la hora de hacer suposiciones, pero viendo los resultados obtenidos, parece que si la tasa es pequeña si existe solución se tarda ligeramente menos en encontrarla que si la tasa es más alta. Mientras que si no existe el algoritmo llegará al máximo de iteraciones con la tasa de emigración mas alta. Esa diferencia de tiempo para encontrar la solución y para parar el algoritmo en caso de no encontrar solución, es pequeña por lo que considero que lo mejor es una tasa pequeña, ya que usaremos estos algoritmos principalmente en problemas en los que existe una solución y, en principio, el algoritmo debería encontrarla, de esta manera ganaremos un poco de tiempo y, en caso de no existir solución (o el algoritmo no encontrarla), perderemos relativamente poco tiempo. Además, aunque no lo he comprobado, pero si la tasa de emigración es extremadamente grande quizás los tiempos para encontrar la solución entre los dos algoritmos si se distancia mucho.

Por último, solo queda comparar la implementación del algoritmo genético híbrido con el modelo de islas (tras el razonamiento expuesto antes, usaré el de tasa de emigración baja). Si observamos los tiempos, vemos que realmente los dos funcionan en un tiempo muy parecido y visto que la aleatoriedad distorsiona un poco los tiempos de ejecución, me arriesgaría a decir que más o menos tardan lo mismo. Un tema interesante habría sido abordar el modelo de islas con programación por hilos, ya que creo que los tiempos de ejecución en ese caso si que habrían diferido muchísimo. Aún así, son muy buenos en los dos casos y prácticamente iguales, lo que no es de extrañar, ya que los algoritmos genéticos que operan detrás son parecidos (recordemos que el modelo de islas tiene dos algoritmos genéticos que son una modificación propia del híbrido genético).

12 http://www.info.univ-angers.fr/pub/porumbel/graphs/flat1000_50_0.col

13 <http://www.info.univ-angers.fr/pub/porumbel/graphs/>

7.0 – Bibliografía y webgrafía.

- Alonso Jiménez, José A., Manual de Lisp, Dpto. de Algebra, Computación, Geometría y Topología Universidad de Sevilla.
- **DIMACS Graphs: Benchmark Instances and Best Upper Bounds.**
- <http://chiosoco.blogspot.com.es/2010/05/blog-post.html> (consultado por última vez 15 – 08 – 2013).
- <http://www.cliki.net/> (consultado por última vez 15 – 08 – 2013).
- <http://geneura.ugr.es/~jmerelo/ie/ags.htm> (consultado por última vez 15 – 08 – 2013).
- <http://eddyalfaro.galeon.com/geneticos.html> (consultado por última vez 15 – 08 – 2013).
- M. M. Hindi, R. V. Yampolskiy "Genetic Algorithm Applied to the Graph Coloring Problem" *Proceedings of the 23rd Midwest Artificial Intelligence and Cognitive Science Conference* (2012), págs 60-66.