Universidad de la República
Facultad de Ingeniería

Proyecto final - Ingeniería en Computación

Algoritmos Genéticos Paralelos para el Problema General de Steiner en Grafos.

Santiago Arraga
arraga@hc.edu.uy

Miguel Aroztegui
marozteg@yahoo.com

Tutores:
Dr. Ing. Héctor Cancela
Ing. Sergio Nesmachnow

Julio 2002

Resumen

Este trabajo presenta un algoritmo genético paralelo para encontrar soluciones aproximadas al Problema de Steiner Generalizado (GSP). El problema esta vinculado con la construcción de una red confiable y de mínimo costo. En la construcción de estas redes se contraponen dos objetivos: minimización del costo total de la red y la maximización de su confiabilidad.

Existen muchas aplicaciones al modelo GSP pero se conocen pocas heurísticas que resuelvan el problema en forma eficiente dado que es NP-completo. Este proyecto aporta una nueva forma de encontrar soluciones aproximadas al GSP para el caso mas general de grafos utilizando un algoritmo genético.

El punto esencial en la construcción del GA fue la representación de una solución factible. Se presentan dos representaciones y se comparan ventajas y desventajas de cada una.

También se presentan distintos modelos de paralelismo en algoritmos genéticos y se provee una implementación para dos de ellos. Se presenta una comparación de los modelo implementados entres si con el algoritmo genético sin paralelizar.

Palabras clave: algoritmos genéticos, paralelismo, problema general de Steiner en grafos, metaheurísticas, migración, vecindad.

1 Introducción

1.1 Descripción del proyecto.

"Algoritmos genéticos paralelos para el problema generalizado de Steiner en grafos" es un proyecto de Taller V realizado en el Departamento de Investigación Operativa del Instituto de Computación de la Facultad de Ingeniería con la supervisión del Dr. Ing. Héctor Cancela y el Ing. Sergio Nesmachnow.

El problema generalizado de Steiner en grafos es un problema de optimización NP-completo. Es de interés encontrar heurísticas que den buenas soluciones al mismo; una vez encontradas y probadas las mismas, un desarrollo lógico posterior es el tratar de hacerlas más rápidas. El primer objetivo de este proyecto fue el estudio, diseño e implementación de una heurística particular para este problema, basada en algoritmos genéticos.

Este objetivo, para una heurística determinada, se logra aumentando la capacidad de procesamiento. Esto se puede hacer de dos maneras no excluyentes; la primera consiste en aumentar la velocidad del hardware utilizado. La segunda consite en distribuir el problema entre distintos procesadores, paralelizándolo. El segundo objetivo de este proyecto fue el estudio y comparación de algunos de estos mecanismos.

El proyecto se realizó durante el periodo de marzo de 2001 – junio de 2002; el código completo del sistema, junto con los casos de prueba, y los resultados obtenidos, se encuentran disponibles en CD.

1.2 Organización del informe

1.2.1 Fundamentos teóricos

El problema generalizado de Steiner en grafo es presentado en el segundo capítulo, junto un ejemplo y una formalización del mismo. A continuación, en el tercer capítulo introducimos los conceptos básicos de algoritmos genéticos y distintas formas de paralelizarlo.

1.2.2 Diseño e implementación del sistema.

En el capítulo número cuatro se discute el diseño y distintas estructuras para representar un algoritmo genético. Tiene como grandes subtemas la elección de una estructura correcta para representar una solución al problema y la implementación de los modelos de paralelismo elegidos.

En el siguiente capítulo se presentan las herramientas utilizadas para implementar el sistema diseñado, incluyendo hardware, limitaciones impuestas al sistema por el mismo, y librerías de paralelismo utilizadas.

Finalmente, en el capítulo número 6 se realiza la validación del sistema implementado, utilizando los mismos grafos de prueba utilizados en Robledo[6].

1.2.3 Pruebas

En el capítulo siete comparamos la performance y bondad de un algoritmo genético paralelo frente a un algoritmo genético equivalente monolítico.

A continuación, en el octavo capítulo, presentamos una evaluación de los distintos parámetros que afectan al modelo de migración. En el noveno capítulo hacemos lo mismo para el modelo de vecindad.

En el décimo capítulo comparamos los modelos paralelos entre sí, y finalmente con el algoritmo monolítico.

1.2.4 Conclusiones

Resumimos los resultados obtenidos en los capítulos anteriores, y presentamos posibles direcciones de investigación futura en el capítulo número once.

La bibliografía es citada en el capítulo doce.

Los detalles de implementación y de las pruebas se dan en el capítulo 13 (Apéndices).

2 Descripción del problema Generalized Steiner Network Problem (GSNP)

Supongamos que somos una empresa de telecomunicaciones que provee de conexiones WAN (Wide Area Network) entre ciudades a una variedad de clientes. Cada ciudad posee una central que maneja los enlaces de y hacia la misma. Cada conexión entre un par de ciudades se compone de un conjunto de enlaces.

Cada cliente tiene requerimientos específicos de conexión; mientras que algunos quieren conectar un par de ciudades mediante un solo enlace, otros quieren que sus conexiones sean resistentes a fallas con un grado de seguridad determinado.

Una conexión entre un par de ciudades puede ir a través de un conjunto de enlaces, de central en central hasta llegar a conectar ambas ciudades; cuando se detecta una falla en la conexión, no se sabe instantáneamente cuál enlace es el fallido, lo cual genera un tiempo de pérdida de servicio inaceptable para los clientes.

A fin de asegurar la conectividad ininterrumpida para aquellos clientes que así lo deseen, se pueden proveer conexiones redundantes. Esto implica que:

- Existen *r* conexiones entre un par de ciudades dado.
- Si un enlace participa en una conexión c_i, con $i = 1..r$, no puede participar en ninguna otra.

La empresa puede tener centrales de conexiones en ciudades donde no haya ningún cliente, y un enlace puede ser compartido por cualquier cantidad de clientes.

Ejemplo:

En este ejemplo, poseemos centrales en las siguientes ciudades: Montevideo, Las Piedras, San José, Colonia, Nueva Palmira, Fray Bentos, Mercedes, Paysandú, Salto, Artigas, Rivera, Melo, Treinta y Tres, Minas, Piriápolis y Punta del Este.

Asimismo, se muestran los costos aproximados de mantener un enlace entre las sucursales que lo permitan. En este ejemplo, se han tomado costos basados en la distancia euclidiana entre las sucursales.

Tenemos tres empresas que contratan nuestros servicios:

- Banco de Tupambaé: Tiene sucursales en Montevideo, Mercedes, Salto, Minas y Rivera. Exige que la sucursal de Montevideo esté conectada por conexiones triplemente redundantes (r=3) al resto, y quiere conexiones doblemente redundantes (r=2) entre las sucursales restantes.

- Cadena de supermercados Litoral: Posee sucursales en Colonia, Nueva Palmira, Fray Bentos, Paysandú y Salto. Quiere conexiones sencillas entre sus sucursales.

- FM “Arena y Olas”: Posee estaciones repetidoras en Montevideo, Piriápolis y Punta del Este. Debido a las exigencias de estar al aire permanentemente y en tiempo real, quiere conexiones doblemente redundantes (r=2) entre estas ciudades.

Nuestra tarea es instalar una red de enlaces de tal manera que se cumplan todos los requerimientos de conectividad de los clientes, minimizando el costo total de la misma. Dado que el costo de instalar un enlace de una central a otra puede variar según factores no controlables, debemos elegir qué enlaces instalar, de forma de reducir el costo total.

Una posible solución al problema anterior, "a ojo de buen cubero", sería la siguiente:

Formalizamos el problema de la siguiente manera: dado un grafo $G(V,E)$ no dirigido con costos no negativos asociado a sus aristas, y un conjunto de requerimientos $R(v_i, v_j, r_{ij})$, con $v_i, v_j \in V$ y $r_{ij} \geq 0$, diremos que un subgrafo G' (V',E') de G es *factible* si $\forall (v_i, v_j, r_{ij}) \in R$

- $v_i \in V', v_j \in V'$
- $\exists$ r_{ij} caminos disjuntos en aristas entre v_i y v_j.

Decimos que un subgrafo G'' factible es *solución del problema generalizado de Steiner* si $\forall$ G' factible, el costo total de G' es mayor o igual que el costo de G''.

Denominamos a los vértices v_i como *terminales*. Los vértices que no participan en niguna restricción se denomina *vértices de Steiner*.

El problema presentando como ejemplo se puede formalizar de la siguiente manera:

1. Cada sucursal se representa como un vértice del grafo G a analizar.

2. Cada posible enlace entre pares de sucursales se representa como una arista entre sus vértices correspondientes.
3. Para cada par de sucursales representadas por los vértices v_i y v_j, agregamos una tripla (v_i, v_j, r_{ij}) si tenemos requerimientos de conexión entre estas sucursales por parte de algún cliente. Hacemos r_{ij} igual al mayor requerimiento de redundancia pedido por los clientes.

Central	Vértice
Montevideo	0
Las Piedras	1
San José	2
Colonia	3
Nueva Palmira	4
Fray Bentos	5
Mercedes	6
Paysandú	7
Salto	8
Artigas	9
Rivera	10
Melo	11
Treinta y Tres	12
Minas	13
Piriápolis	14
Punta del Este	15

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	-	1	-	3	-	-	-	-	-	-	8	-	-	2	2	-
1	1	-	2	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	2	-	-	-	-	4	-	-	-	-	-	-	-	-	-
3	3	-	-	-	2	-	-	-	-	-	-	-	-	-	-	-
4	-	-	-	2	-	2	-	-	-	-	-	-	-	-	-	-
5	-	-	-	-	2	-	1	-	-	-	-	-	-	-	-	-
6	-	-	4	-	-	1	-	3	-	-	7	-	-	-	-	-
7	-	-	-	-	-	-	3	-	3	-	-	-	9	-	-	-
8	-	-	-	-	-	-	-	3	-	4	5	-	-	8	-	-
9	-	-	-	-	-	-	-	-	4	-	2	-	-	-	-	-
10	8	-	-	-	-	-	7	-	5	2	-	5	-	-	-	-
11	-	-	-	-	-	-	-	-	-	-	5	-	3	-	-	-
12	-	-	-	-	-	-	-	9	-	-	-	3	-	4	-	-
13	2	-	-	-	-	-	-	-	8	-	-	-	4	-	1	1
14	2	-	-	-	-	-	-	-	-	-	-	-	-	1	-	2
15	-	-	-	-	-	-	-	-	-	-	-	-	-	1	2	-

v_i	v_j	r_{ij}
0	6	3
0	8	3
0	10	3
0	13	3
6	8	2
6	10	2
6	13	2
8	10	2
8	13	2
10	13	2
3	4	1
3	7	1
3	8	1
4	7	1
4	8	1
7	8	1
0	14	2
0	15	2
14	15	2

Las dos tablas presentadas arriba muestran la correspondencia entre ciudades y vértices, y otra mostrando los costos asociados a las aristas entre cada par de vértices.

A la izquierda se muestra la tabla de restricciones que debe de cumplir el grafo. Notar que las restricciones son simétricas; es decir, que si es necesario que existan r_{ij} conexiones entre un par de vértices v_i y v_j, deben existir r_{ij} caminos disjuntos respecto a aristas entre v_j y v_i.

Finalmente, a la derecha se muestra la mejor solución encontrada hasta el momento.

Este problema, que fue originalmente formulado por Krarup [1], es también llamado *síntesis de redes multiterminales* por Chien [2] y Gomory y Hu [3] (citados por Klein y Ravi [4]). Asimismo, es referido como el problema de diseñar *redes robustas* (*survivable networks*) de costo mínimo por Steiglitz, Weiner y Kleitman [5] (también citados por Klein y Ravi [4]).

El problema es NP-Completo [1], por lo que es de sumo interés encontrar heurísticas que encuentren buenas soluciones al mismo.

Está relacionado con el problema del árbol de Steiner sobre grafos (*Steiner Tree problem*), el cual está definido como sigue: Dado un grafo *G*(*V,E*) con una función de costo $c : E \to N$ y un subconjunto de vértices $X \subseteq V$, el objetivo es encontrar un árbol de mínimo costo que incluya a todos los vértices de *X*. Existe un volumen de trabajo mucho mayor sobre esta variante, con diversas heurísticas y aproximaciones disponibles. Winter [8] realiza un relevamiento de publicaciones sobre el GNSP.

Robledo[6] propone una heurística basada en *Ant Systems* para este problema. Charikar, Chekuri *et al* [7] proponen una serie de algoritmos de aproximación para este tipo de problemas en grafos *dirigidos*.

Klein y Ravi [4] encuentran un algoritmo en tiempo polinomial para encontrar un subgrafo *G'* factible y demuestran que su costo está a no más de $(2-\frac{2}{k})\lceil \log_2(r_{max}+1) \rceil$ del costo de la solución óptima. Esto da una primera idea de la bondad o no de una heurística, puesto que establece una cota inferior de performance que debe cumplir.

3 Introducción a los algoritmos genéticos

Los algoritmos genéticos son algoritmos de búsqueda basados en la idea de codificar una posible solución al problema en un vector de componentes. Esta idea es análoga a la codificación de información genética en biología.

Las propiedades destacables de los algoritmos genéticos son robustez frente a diferentes tipos de problemas, simplicidad de las estructuras y operaciones involucradas [9].

A continuación, se explican algunos conceptos heredados de la biología usados en este trabajo. Al final del capítulo uniremos estos términos dando una idea clara de que se trata un algoritmo genético simple aplicado a un problema de búsqueda genérico.

3.1 Definiciones

3.1.1 Problema, espacio de búsqueda, solución [no] factible.

Usaremos la siguiente definición genérica para nuestro *problema*:

Se desea encontrar una solución factible X en el espacio de soluciones U tal que se maximiza la función objetivo F:U→ℜ.

Al espacio de soluciones *U* también lo llamaremos ***espacio de búsqueda***.

Las restricciones o condiciones del problema particionan el espacio de búsqueda *U* en dos conjuntos de soluciones: ***factibles*** y ***no factibles***. Las soluciones factibles son puntos que cumplen las condiciones del problema.

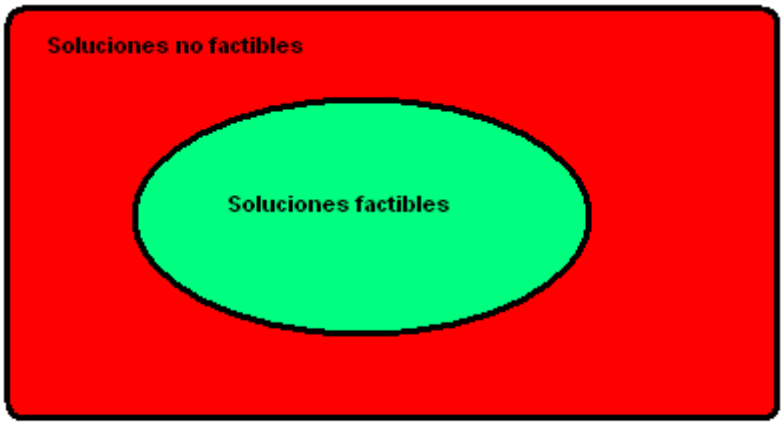


Figura 3.1

3.1.2 Cromosoma, Gen y Alelo.

La estructura básica en la cual trabaja el algoritmo genético es el ***cromosoma***.

En algoritmos genéticos un cromosoma es un vector de ***genes***. En biología un gen describe determinado caracter de un individuo, por ejemplo el color de los ojos. El valor asignado a un gen, por ejemplo, color azul, se denomina ***alelo***. Un gen no tiene porque estar en la misma posición del vector. A los efectos de este trabajo consideraremos que los genes están en posiciones fijas del vector.

Cromosoma, la estructura básica de un algoritmo genético.

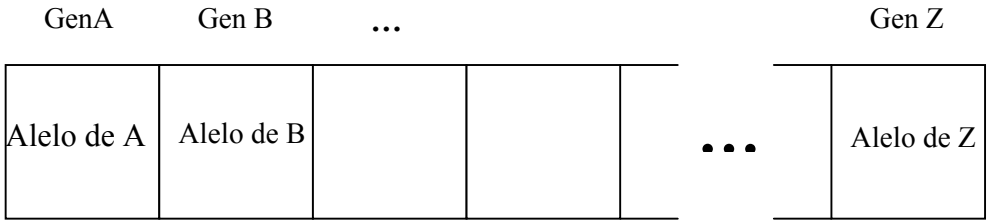


Figura 3.2

La estructura de datos de un alelo es arbitraria y elegida convenientemente para la resolución del problema en cuestión. La estructura más simple de un alelo es la de un bit y es la que se utilizará en este capítulo.

3.1.3 Genotipo/ fenotipo, codificación/decodificación.

En Biología, ***genotipo*** es un conjunto de cromosomas que definen las características de un individuo.

El genotipo sometido a un medio ambiente se llama ***fenotipo***. Es decir, el fenotipo es el individuo tal cual lo vemos, afectado por el medio ambiente. Como ejemplo podemos citar unas plantas que poseen el mismo genotipo pero según estén sumergidas, parcialmente sumergidas o sobre superficie seca tendrán distintos fenotipos. Este y otros ejemplos de la naturaleza pueden verse en [10].

En términos de los algoritmos genéticos el genotipo generalmente esta constituido por un único cromosoma. Es decir un individuo esta constituido por un único cromosoma. En este documento usaremos los términos ***genotipo***, ***cromosoma*** e ***individuo*** indistintamente.

En algoritmos genéticos, **el fenotipo representa un punto del espacio de soluciones *U***. Esto es lo que el usuario del algoritmo entiende, sabe como interpretarlo.

Como habíamos mencionado, el algoritmo genético trabaja sobre cromosomas únicamente. Por lo tanto se debe definir una función de ***codificación*** (***Cod***) sobre los puntos del espacio de soluciones:

$$Cod : U \rightarrow cromosoma$$

que mapea todo punto del espacio de soluciones en un genotipo. Idénticamente la inversa de *Cod* (*Decod*) obtiene un fenotipo dado un genotipo.

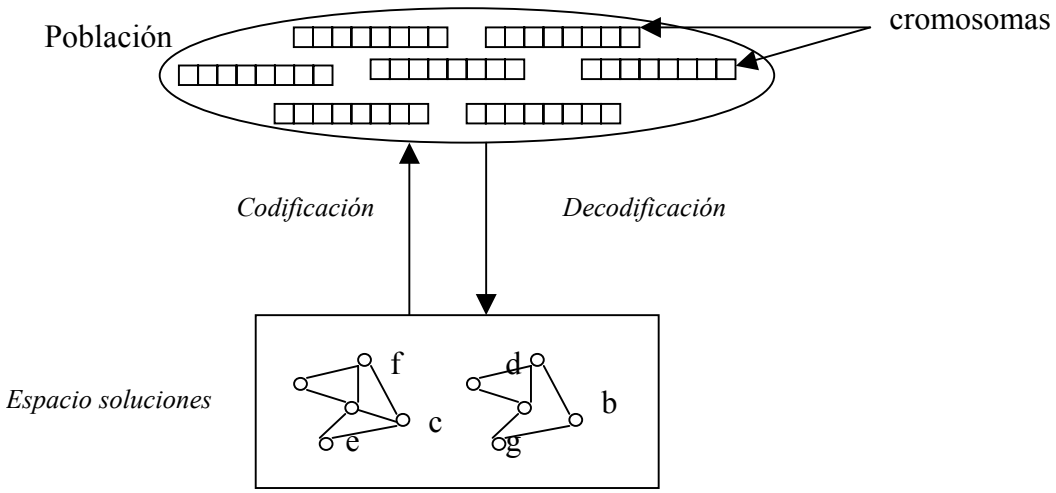


Figura 3.3

3.1.4 Población

El algoritmo genético mantiene un conjunto finito de cromosomas llamado ***población***. Este conjunto evolucionará de generación en generación a medida que avance la ejecución del algoritmo.

3.1.5 Fitness

Todo cromosoma tiene un valor asociado de ***fitness*** (adaptación, grado de bondad de la solución codificada en el vector), el cual corresponde, a la aptitud del individuo dentro de la población. El cálculo del fitness se hace sobre el fenotipo correspondiente al cromosoma. En resumen, *fitness* es una función del mismo tipo que la función objetivo *F*.

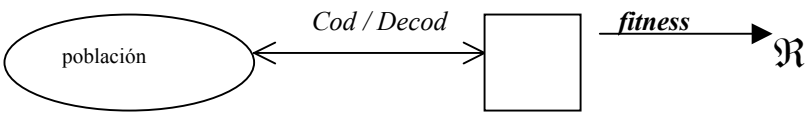


Figura 3.4

La evolución de los cromosomas de una población se basará en sus fitness correspondientes. El algoritmo genético trata de simular la evolución de las poblaciones según la teoría evolutiva darwiniana: individuos con mayor fitness sobrevivirán y participarán en la reproducción de nuevos individuos de la nueva generación.

En términos de nuestro problema genérico, debemos expresar el fitness en base a la función objetivo ***F***. Para fijar ideas consideramos:

$$\text{fitness} = F$$

3.1.6 Operadores: reproducción, cruzamiento y mutación.

Un algoritmo genético [9] aplica repetitivamente y en este orden los siguientes operadores sobre cromosomas de la población:

1) Selección
2) Cruzamiento
3) Mutación

El resultado de la aplicación de 1), 2) y 3) es una nueva generación de cromosomas que integrarán la nueva población sobre la cual se volverá aplicar los operadores.

A continuación se detalla cada uno de estos operadores.

Selección

Esta operación consta en la selección (basado en la fitness) y clonación de cromosomas de la población. Estos cromosomas clonados pasan a integrar un conjunto de individuos a los cuales se aplicará el cruzamiento y la mutación. A este conjunto lo llamaremos conjunto de apareamiento (mating pool).

Se debe definir por lo tanto una operación de selección:

$$\textit{Select : Población} \rightarrow \textit{cromosoma}$$

que decide cual cromosoma pasará a integrar el conjunto de apareamiento.

El mecanismo de Select se puede simular con una ruleta [9] en la cual todo cromosoma de la población ocupa un área de la misma. Pongamos un ejemplo, supongamos que la población esta integrada por cinco individuos. A continuación están sus características en la tabla y la ruleta asociada.

Cromosoma	Fitness
A	40
B	30
C	15
D	10
E	5

Tabla 3.1

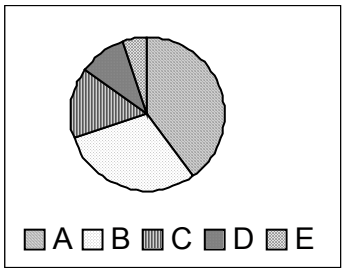


Figura 3.5

Para realizar el select basta con hacer girar la ruleta, la misma se detendrá en un 40% sobre el cromosoma A, un 30% sobre el B, … y un 5% sobre E.

Existen otras formas de implementar la operación de selección. Algunas de ellas son:

- Torneo: Se elige un par de cromosomas con probabilidad uniforme de la población original. Se comparan las fitness de ambos, y se ingresa el que tiene mejor fitness a la nueva población.

  Este proceso se repite hasta que la nueva población tiene la cantidad deseada de cromosomas.

- Ranking: Se toman los mejores *k* cromosmas de la población, y se utiliza esa subpoblación (*mating pool*) para el resto de las operaciones genéticas.

Cruzamiento

Esta operación elige al azar dos cromosomas padres del conjunto de apareamiento y se obtiene como resultado dos nuevos cromosomas hijos constituidos con la información genética de los padres. Los hijos se guardan en la nueva población en construcción.

Ilustraremos tres forma de realizar el cruzamiento:

- en un punto
- en dos puntos
- uniforme

El cruzamiento en un punto consiste en la elección de una posición del vector al azar. Supongamos que el vector tiene N alelos, entonces se deberá sortear un valor entre 2 y N. Supongamos que la posición elegida es K. A continuación se ilustran los dos cromosomas antes y después del cruzamiento.

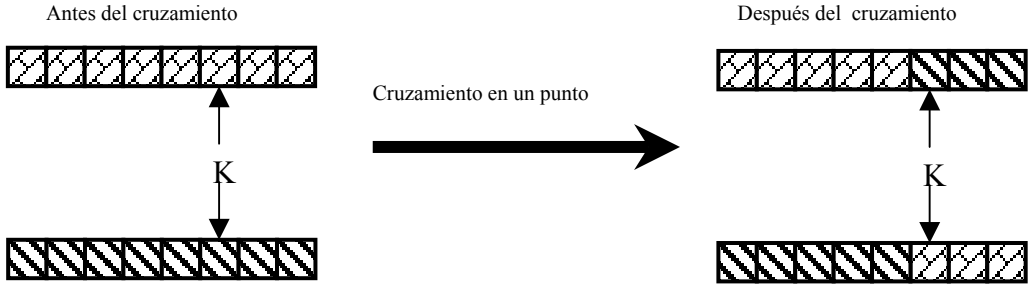


Figura 3.6

En el cruzamiento en dos puntos se eligen dos posiciones K y J (K<J) en el rango de 2 a N y se intercambian los alelos entre K y J como se ilustra a continuación.

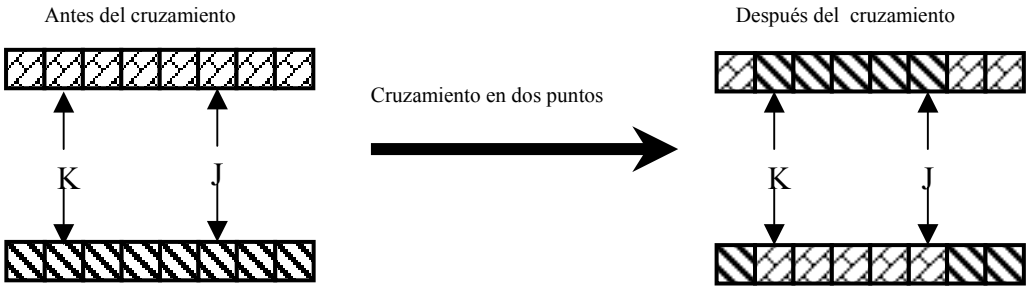


Figura 3.7

Por último, el cruzamiento uniforme consiste en decidir aleatoriamente para cada posición de 1 a N si hay o no intercambio de alelos. A continuación se ilustra el cruzamiento uniforme en K, J y L.

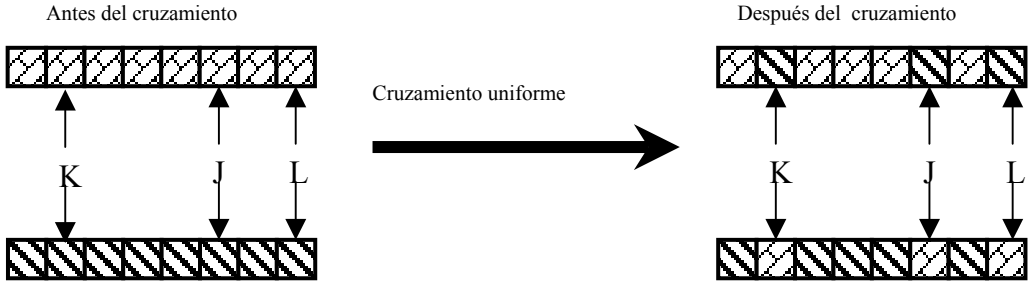


Figura 3.8

Mutación

Es la alteración del valor de un alelo. Para nuestro caso (alelo = bit), simplemente invertimos el bit asignado en el alelo.

3.2 Esquema general de un algoritmo genético

A continuación esta un pseudocódigo del algoritmo genético sencillo. En mismo menciona conceptos descriptos en los puntos anteriores de este capítulo e introduce otros nuevos que se explicarán mas adelante en este punto.

1. Generar aleatoriamente población ***pop*** con ***SIZEPOP*** cromosomas [factibles].
2. Generar población ***newPop*** con 0 cromosomas.
3. Mientras (no se cumpla la ***condición de parada*** del algoritmo) hacer
    4. Mientras (***newPop*** tenga menos que ***SIZEPOP*** cromosomas) hacer
        5. ***select*** de dos cromosomas c1 y c2 de ***pop*** en función de su fitness.
        6. ***cruzamiento*** de c1 y c2 con probabilidad ***PCROSS***. Los cromosomas cruzados los llamaremos c11 y c22.
        7. ***mutación*** de los genes de c11 y c22 con probabilidad ***PMUT***. Los cromosomas mutados los llamaremos c111 y c222.
        8. [Si c111 es factible,] se inserta c111 en ***newPop***. [Si c222 es factible,] se inserta c222 en ***newPop***.
        9. ***pop*** = ***newPop***.

10. El resultado del algoritmo esta en ***pop***. Para obtener la menor solución hay que decodificar el mejor cromosoma encontrado en ***pop***.

Observaciones:

- No se utiliza un conjunto de apareamiento (mating pool) como se menciona en el punto 2.1.6 (Selección). Simplemente se va construyendo la nueva generación ***newPop*** de a pares de cromosomas.
- Por lo general no se utiliza la función de codificación del espacio de búsqueda explícitamente. En otras palabras, cuando se genera la población inicial en el punto 1 **no** se toman puntos del espacio de soluciones aleatoriamente y se codifican los mismos.
- Definimos cromosomas factibles como aquellos cuya decodificación es factible. Es posible hacer cromosomas tales que el cruzamiento y la mutación pueden devolver cromosomas no factibles. En tales casos existen dos posibilidades de tratar con estos cromosomas. La primera es simplemente considerarlos con una fitness muy mala e insertarlos en la nueva generación. Otra posibilidad es descartarlos y mantener únicamente poblaciones de cromosomas factibles. Uno de estos tratamientos esta entre paréntesis rectos [].
- Es posible optimizar esta versión del algoritmo genético. En particular el cruzamiento y la mutación se pueden hacer simultáneamente en una única función. En esta versión se separaron estos operadores por claridad.

La ***condición de parada*** es la combinación de condiciones que al cumplirse alguna detienen el algoritmo. Típicamente estas son:

1) Número máximo de generaciones: una generación es una población construida a partir de la anterior, esto ocurre desde los pasos 4 a 9 del algoritmo. Para esta condición, el algoritmo se detiene cuando alcanza el máximo número de generaciones.
2) Intervención del usuario: el usuario puede querer detener en determinado instante la ejecución del algoritmo.
3) Tiempo: luego de transcurrido determinado periodo de tiempo desde el comienzo de la ejecución el algoritmo termina.
4) Convergencia: la última población generada no ha mejorado lo suficiente en comparación con alguna generación anterior por lo tanto puede tener poco sentido seguir generando nuevas poblaciones.

A continuación se describen parámetros destacables en los algoritmos genéticos. Los valores recomendados para los mismos están entre paréntesis [].

- ***SIZEPOP***: Número de cromosomas de la población [20 a 30. En otros casos es de 50 a 100. Investigadores dicen que los valores dependen de la codificación y largo del cromosoma].
- ***PCROSS***: Probabilidad de cruzamiento entre dos cromosomas [80%-95%].
- ***PMUT***: Probabilidad de mutación de un gen [0.5% a 1%].

Los valores para los parámetros anteriores se extrajeron de una pagina web de Internet [11].

3.3 Paralelismo

3.3.1 Por qué paralelizar?

El aumento de la capacidad de procesamiento de cualquier algoritmo puede venir por dos caminos distintos:

- Aumento en la velocidad de la computadora donde se ejecute el algoritmo.
- Paralelización del algoritmo, ejecutándolo al mismo tiempo en múltiples computadoras.

Si bien hay algoritmos para los cuales la segunda opción es sumamente ineficaz, los algoritmos genéticos no sólo no caen dentro de esta clase, sino que son altamente paralelizables.

Para ver esto, examinemos una corrida típica de un algoritmo genético. Al inicio, se crea una población de individuos. En cada generación se evalúa la factibilidad de cada individuo, la cual típicamente es una propiedad independiente del resto de los individuos de la población.

Luego de esta evaluación, se selecciona un subconjunto de la población de acuerdo a una función de probabilidad directamente relacionada con la factibilidad de cada individuo. Esta selección se realiza de forma independiente de los individuos seleccionadas anteriormente. Formalmente, si el evento C_i significa "seleccionar el cromosoma i-ésimo de la población", $P(C_i) = P(C_i / C_j)$ con $i, j = 1...N$. Al no depender la selección de cada individuo de los ya seleccionados, ésta se puede hacer en forma paralela.

Los individuos de cada generación posterior a la inicial se obtienen de la generación anterior mediante la aplicación de un conjunto de operaciones genéticas a los individuos perteneciente a la subpoblación seleccionada en el paso anterior. Esta operaciones son típicamente las siguientes:

1. Reproducción: Se inserta una copia de un individuo.
   $c' = c$.

2. Cruzamiento: Se toman dos individuos de la subpoblación seleccionada, y se generan dos nuevos individuos tomando trozos del material genético de los originales, de acuerdo a diversos criterios (cruzamiento en un punto, en dos puntos, uniforme) explicados anteriormente.
   $(c_1', c_2') = c_1 \times c_2$

3. Mutación: Se toma un individuo de la subpoblación seleccionada, y se cambia al azar parte de su material genético.
   $c' = \perp c$

Como podemos ver, cada una de estas operaciones involucra como máximo dos individuos antecesores, no dependiendo de ningún otro factor. Esto implica que cada una de ellas se puede hacer en paralelo.

3.3.2 Niveles de paralelismo

En la mayoría de las ejecuciones de un algoritmo paralelo, se utiliza relativamente poco tiempo del sistema en las tareas de creación de la población inicial, selección de individuos y ejecución de las operaciones genéticas. En general, el componente principal de la carga computacional es el cálculo de la fitness de cada individuo. Sin embargo, esto puede variar con distintas representaciones del individuo.

Las consideraciones anteriores nos llevan a plantear tres niveles de paralelización, originalmente sugeridos en [12]:

- A nivel de cálculo de factibilidad. Dado de que cada individuo es independiente del resto de la población, es posible calcular su factibilidad por separado.

- A nivel de individuo. Dado de que las operaciones genéticas toman como máximo un par de individuos de la población, y de que no dependen de aplicaciones anteriores de las mismas, es posible realizarlas en paralelo.

- A nivel de población. Al menor nivel de granularidad, es posible tener poblaciones independientes, cada una tratando de resolver el mismo problema y comunicándose entre ellas las mejores soluciones obtenidas al momento.

3.3.3 Paralelización a nivel de cálculo de factibilidad.

En este modelo, tanto la población como todos sus individuos se encuentran contenidos en una sola computadora, la cual corre el algoritmo genético en forma similar a un sistema aislado. Sin embargo, cuando el algoritmo necesita calcular la factibilidad de un individuo, se realiza una invocación remota a un procedimiento apropiado en otra computadora. Estas computadoras tienen como única tarea el escuchar pedidos de cálculo de factibilidad, calcularla, y devolver las respuestas al proceso principal corriendo en la computadora principal.

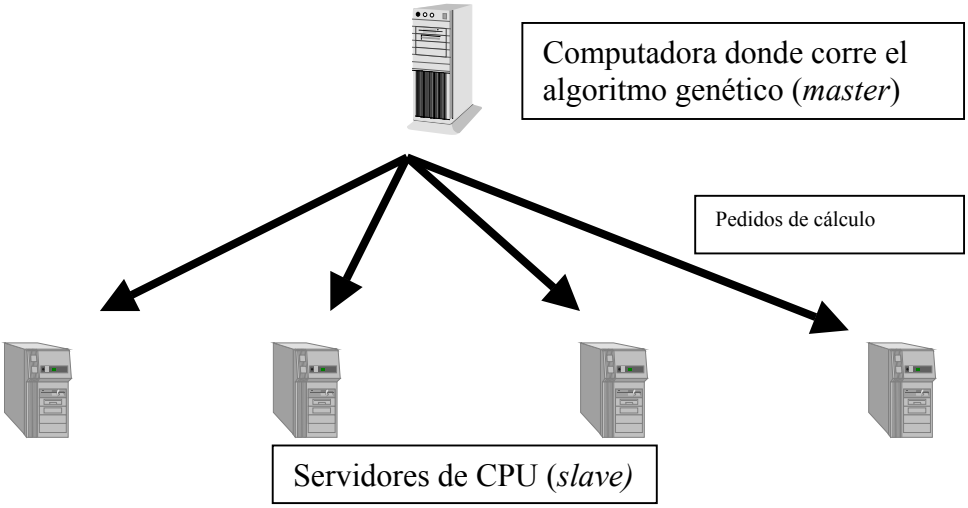


Figura 3.9

Este modelo puede ser apropiado cuando el cálculo de factibilidad es relativamente largo (para contrabalancear los costos de comunicación) o la cantidad de servidores de CPU es aproximadamente igual a la cantidad de individuos de la población [12].

3.3.4 Paralelización a nivel de individuo

Aquí tenemos una única población lógica, pero sus individuos están distribuidos en distintas computadoras. Cada computadora corre en paralelo el mismo algoritmo genético, terminando con la asimetría existente entre los sistemas del modelo anterior.

Cuando se realizan las operaciones genéticas en cada generación, cada computadora puede elegir como antecesor uno de los individuos que están presentes localmente en la misma, o tomarlo de otro sistema. El o los individuos resultantes de la operación se agregan a los individuos locales.

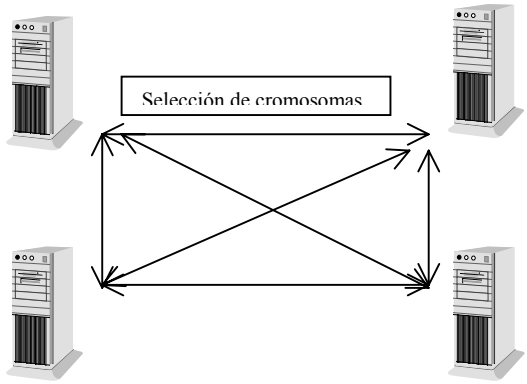


Figura 3.10

3.3.5 Paralelización a nivel de población

En este enfoca, también llamado de "isla" (*island*), existen múltiples poblaciones semi aisladas (*demes*). Cada población es asignada a una computadora distinta, donde corre el mismo algoritmo genético. Tanto el cálculo de factibilidad como la selección de individuos y la aplicación de operaciones genéticas se realizan con individuos locales al sistema.

Al terminar una generación (u otro intervalo), se elige probabilísticamente (basados en fitness) un porcentaje de individuos de cada población para emigrar de un sistema a otro. Estos individuos se agregan a la subpoblación seleccionada para formar la nueva generación de cada sistema.

La paralelización a este nivel tiene como ventaja el hecho de que el tiempo gastado en comunicaciones es pequeño en comparación a los niveles anteriores, dado de que solamente se migra un modesto porcentaje de los individuos en cada generación. Además, cada migración está separada por relativamente largos períodos de tiempo entre una generación y la que sigue.

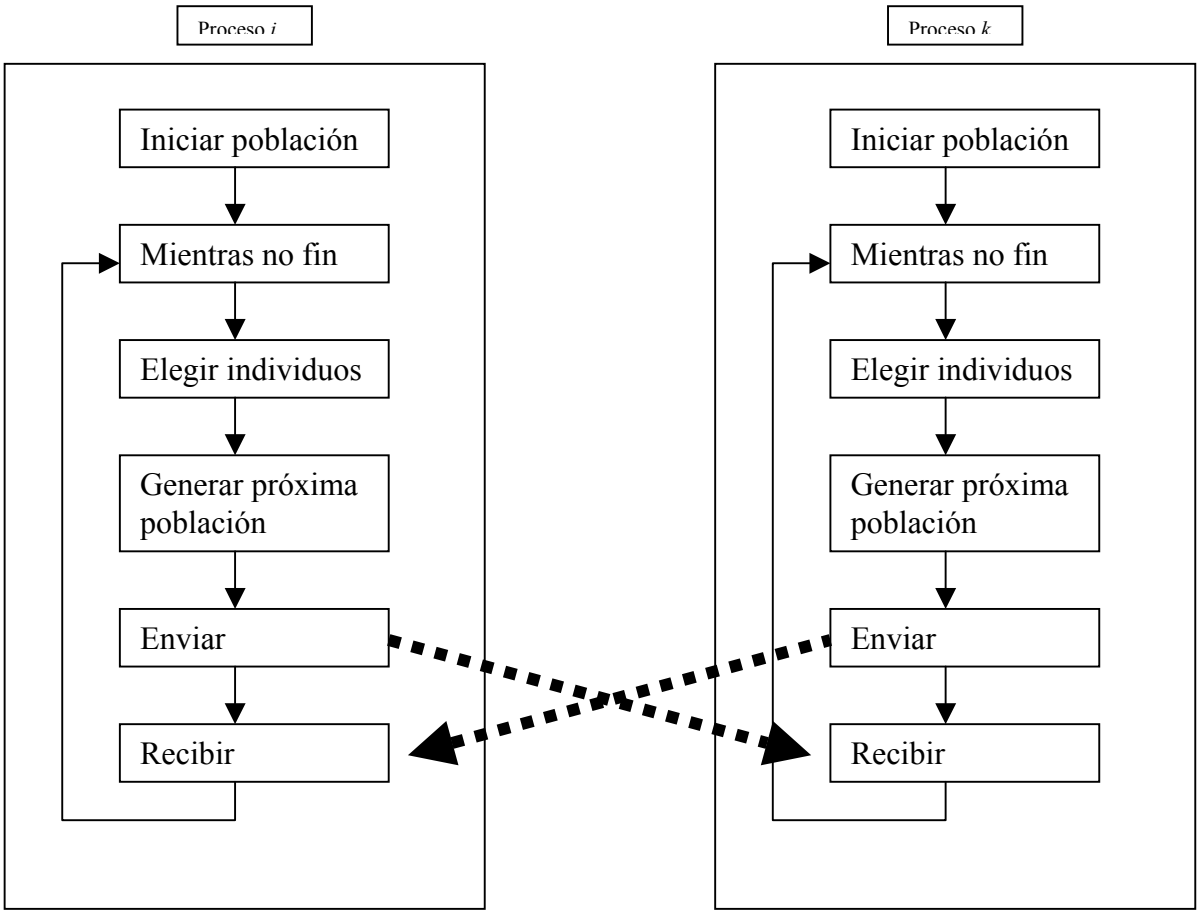


Figura 3.11

Cada nivel presenta una granularidad menor respecto al nivel anterior. Estos niveles *siempre* están presentes en el algoritmo genético general descrito, debido a las características mencionadas en la descripción de cada uno de ellos.

En algunos problemas particulares pueden existir niveles adicionales. Por ejemplo, supongamos que el cálculo de factibilidad involucre la multiplicación de matrices grandes. Es sabido de que se puede paralelizar esta operación en n^2 procesadores, si las matrices son de tamaño $m \times n$ y $n \times j$, por lo que el proceso encargado de calcular la factibilidad (tanto un proceso corriendo el algoritmo genético como un servidor de CPU) podría a su vez paralelizar este cálculo. Sin embargo, esto no es aplicable generalmente.

3.3.6 Modelos de paralelización

Existen varios modelos de paralelización de algoritmos genéticos sugeridos, cada uno de los cuales presenta características de uno o más de los niveles de granularidad discutidos anteriormente.

Schwehm [13] sugiere tres categorías, basadas en el *radio* del operador de selección.

- Global: la población esta unificada y carece de estructura (imaginativamente, una gran ciudad). Todo individuo puede ser elegido para la reproducción.

  Esta categoría es esencialmente paralelización a nivel de individuos.

- Local: la población esta estructurada de acuerdo a una relación de vecindad (podemos hacer una analogía con la vida real y decir de que la población que vive en una ciudad lo hace en *barrios*). Los antecesores de las operaciones genéticas son elegidos preferentemente del mismo barrio o de barrios vecinos. La relación de vecindad está prefijada inicialmente, pero puede variar a lo largo del tiempo. En esta categoría se hace uso de dos niveles distintos de granularidad:

  1. Existe granularidad a nivel de individuos, dado de que se pueden tomar individuos de distintos sistemas (barrios) para formar una nueva generación.

  2. Existe granularidad a nivel de poblaciones. Dado de que los individuos son elegidos dentro de un conjunto limitado de sistemas, se pueden tener conjuntos de barrios disjuntos que forman colectivamente poblaciones lógicas disjuntas.

- Regional: la población es dividida en regiones (*islands* - *demes*), cada una de ellas contenida en un procesador. Las operaciones genéticas se hacen tomando como antecesores a individuos de una misma región. Cada *n* generaciones pueden emigrar individuos de una región a otra. Como vemos, esta categoría es sencillamente paralelización a nivel de poblaciones.

Grefenstette [14], mencionado por Goldberg [9], sugiere los siguientes modelos:

- Master-Slave sincrónico: Un proceso coordinador implementa el algoritmo genético y utiliza un pool de procesos para evaluar funciones. Esencialmente, paralelización a nivel de cálculo de factibilidad.

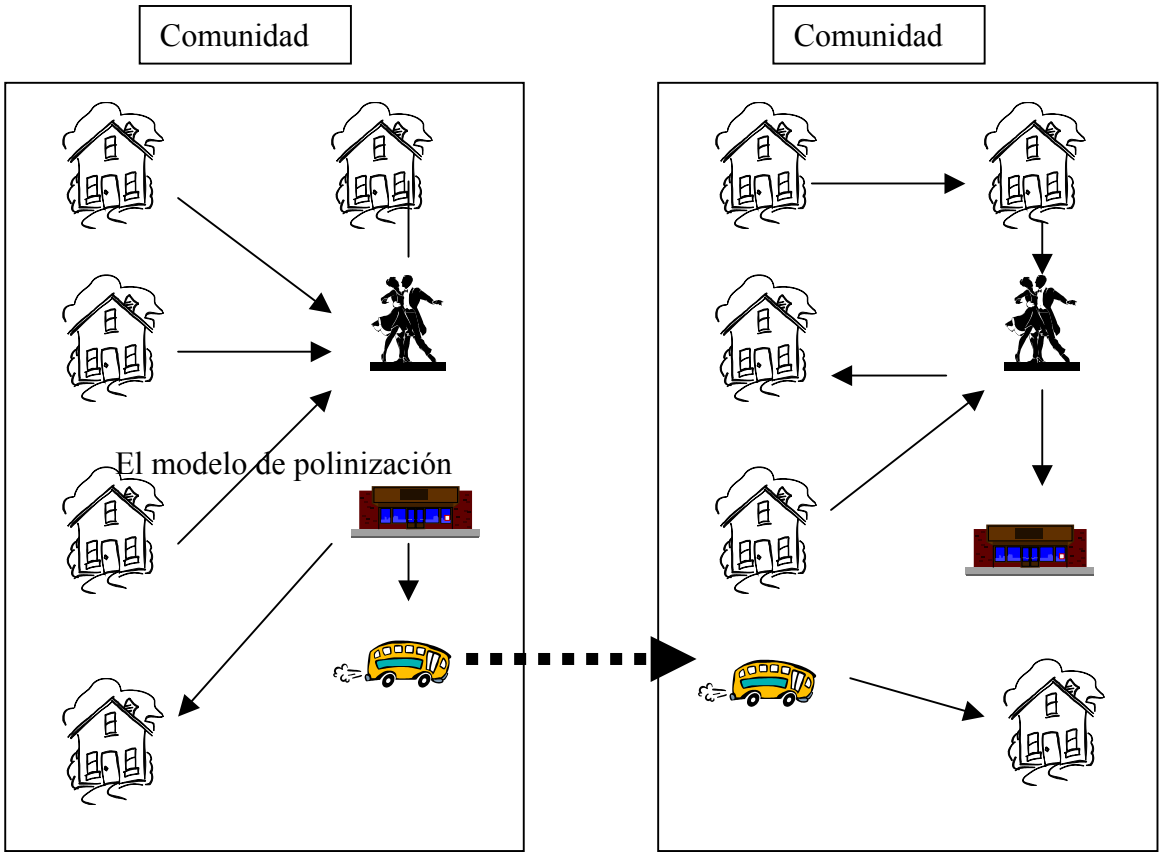
- Master-Slave semisincrónico: un refinamiento del anterior, con la posibilidad de insertar nuevos procesos y reutilizar existentes a medida de que quedan libres.

- Asincrónico concurrente: Un conjunto de procesos, cada uno implementando el algoritmo genético, acceden a una memoria compartida que mantiene la población. Es una forma alternativa de implementar paralelización a nivel de individuos.

- Red: Un conjunto de procesos, sin memoria compartida, cada uno implementando el algoritmo genético trabajan de forma independiente, con la excepción de que los mejores individuos de cada población se comunican vía broadcast al resto de los procesos. Es similar al modelo Regional de Schwehm, valiendo todas las apreciaciones hechas para el mismo.

Goldberg [9] sugiere modelos más complejos y curiosos, los cuales están basados en analogías con la vida real.

El modelo de comunidad consiste en un conjunto de comunidades interconectadas. Cada comunidad consiste en un conjunto de *casas*, en las que los individuos viven. Dos antecesores viviendo en una casa producen un conjunto de nuevos individuos. Estos nuevos individuos son enviados a un *pub*, donde cada uno elige a una pareja. Luego de que se forman las parejas, es tarea de ellas competir por encontrar una casa vía una *agencia inmobiliaria.* Si la comunidad está llena, las parejas pueden elegir mudarse a otra comunidad mediante una *agencia de buses*.



El modelo de *polinización* consiste en una serie de nodos (*plantas*) conectados por una *red de polinización*. Las semillas crecen hasta transformarse en plantas adultas, que generan polen, el cual circula por la red de polinización. Cada arista de la red tiene asociada una probabilidad de transmisión de polen, lo que permite formar subpoblaciones más o menos aisladas una de otra. La selección ocurre localmente, eligiendo a las mejores semillas para tomar el lugar de la planta anterior.

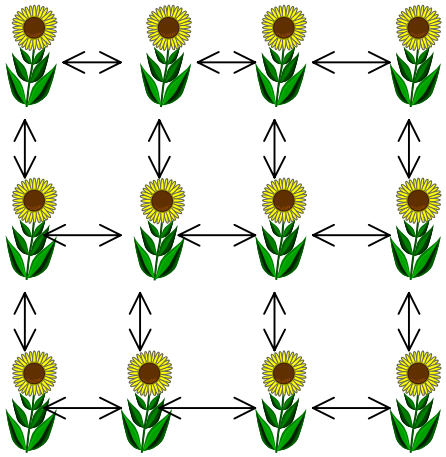


Figura 3.13

3.4 Resumen

Para la construcción de un algoritmo genético simples se deben definir los siguientes puntos:

- Codificación de una solución. Debe tener la forma de vector. Hay que pensar en lo que le puede pasar al cromosoma cuando se realice el cruzamiento y la mutación ya que estas operaciones pueden hacerlo no factible.
- La fitness.
- La selección.
- El cruzamiento.
- La mutación.
- La condición de parada.

4 Diseño del algoritmo genético paralelo

De la literatura estudiada respecto a algoritmos genéticos [9 y 15] se observa la importancia en la elección de la estructura del cromosoma. Describimos dos estructuras posibles. La primera es simple pero luego de un cruzamiento los cromosomas resultantes pueden no ser solución factible del problema. La segunda es más compleja pero sin problemas de factibilidad con los cruzamientos.

4.1 Discusión sobre estructura del cromosoma

4.1.1 Estructura sencilla

Cada cromosoma se representa como un array $[0 \cdots n-1]$ de bits, siendo n el número de aristas. Cada arista del grafo está identificada con un número de 0 a $n-1$. La arista se encuentra presente en el grafo representado por el cromosoma si el bit correspondiente a su posición está seteado.

1	0	1	1	1	0	0	1	1	…	1

Figura 4.1

Operadores:

- Cruzamiento: Se elige un punto de cruzamiento en el rango $[0 \cdots n-1]$ aleatoriamente, y se intercambian los trozos del array.
- Mutación: Se invierte el valor del alelo elegido.

Cálculo de fitness

$\Re = C - \sum_{0}^{n-1} [BIT(i) * COSTO(i)]$, donde C representa el costo del grafo original, $COSTO : \mathrm{N} \to R$ devuelve el costo la arista i-ésima, y la función $BIT : N \to \{0,1\}$ devuelve 1 si el bit correspondiente a la arista i-ésima está seteado y 0 en caso contrario.

Problemas:

Esta representación no garantiza que el grafo representado sea un grafo factible de Steiner. Por ejemplo, luego de un cruzamiento, las soluciones representadas por los antecesores son mezcladas, y no es seguro de que los descendientes sean soluciones válidas. Esto significa que luego de cada cruzamiento o mutación, es necesario verificar la factibilidad del nuevo cromosoma.

Si el cromosoma no es factible, hay dos opciones [15].

- Descartarlo.
- Asignar valores de fitness negativa

Arbitrariamente, dado que no se presentan ventajas de una alternativa sobre la otra en la literatura citada, se eligió descartar los cromosomas no factibles.

Ventajas:

- Implementación sencilla.
- Ausencia de repetición de material genético.
- Utilización mínima de memoria.

4.1.2 Estructura de caminos

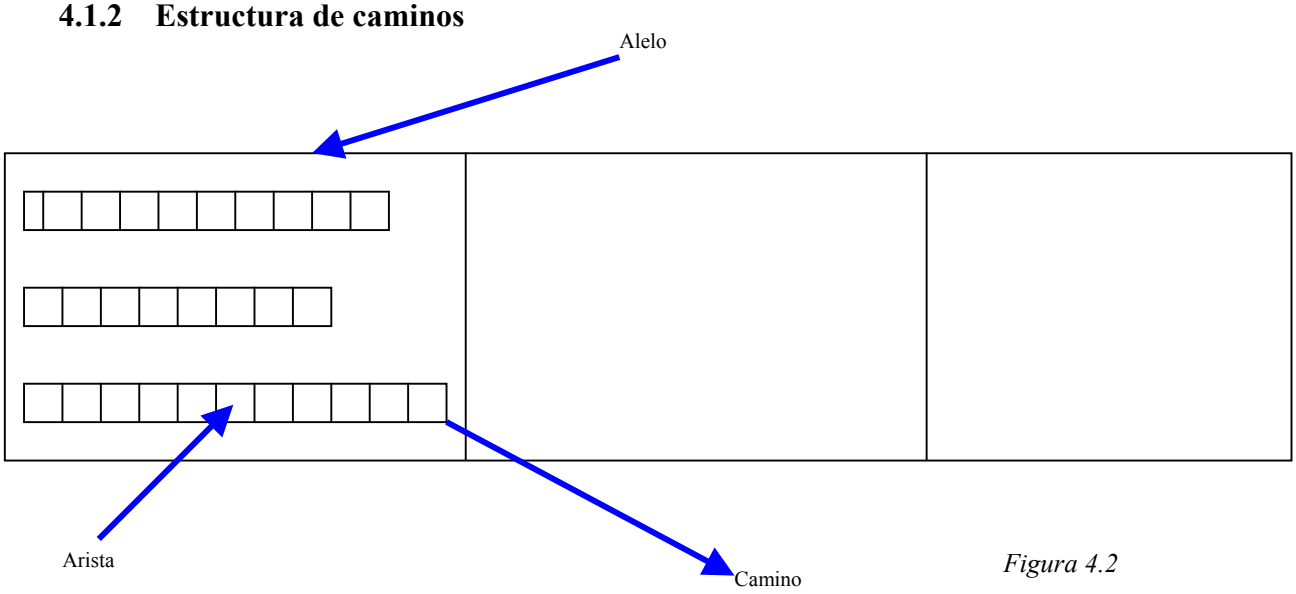


Figura 4.2

El cromosoma se representa como un array $[0 \cdots |R| - 1]$ donde R es el conjunto de restricciones existentes en el grafo.

Cada alelo del cromosoma contiene un conjunto de posibles caminos disjuntos entre el par de vértices *u* y *v* especificados en la restricción, con cardinalidad igual a la restricción.

Cada camino se representa como una lista de aristas.

Operadores

- Cruzamiento: Se elige un punto de cruzamiento en el rango $[0 \cdots |R| - 1]$ aleatoriamente, y se intercambian los trozos del array.
- Mutación: La mutación de un alelo correspondiente a una restricción r_i implica el volver a encontrar k caminos disjuntos respecto a las aristas entre *u* y *v*, siendo *u*, *v* los vértices implicados en la restricción y *k* el número de caminos exigidos en la misma.

Cálculo de Fitness

$\Re = C - \sum_{0}^{n-1} \left[EnCamino(i) * COSTO(i) \right]$, donde C representa el costo del grafo original, $COSTO : \mathrm{N} \rightarrow R$ devuelve el costo la arista i-ésima, y la función $EnCamino : N \rightarrow \{0,1\}$ devuelve 1 si la arista i-ésima pertenece a algùn camino representado en el cromosoma y 0 en caso contrario.

Problemas:

- Repetición de material genético cuando la cantidad de restricciones es grande. A medida que aumenta la misma, hay mayores posibilidades de obtener el grafo de entrada.
- Uso masivo de memoria para representar cada cromosoma.

Ventajas:

- El cruzamiento y la mutación siempre dan en soluciones factibles al problema de Steiner generalizado, minimizando la cantidad de procesamiento necesario, dado que no es necesario verificar la factibilidad de cada cromosoma.

4.1.3 Estructura elegida

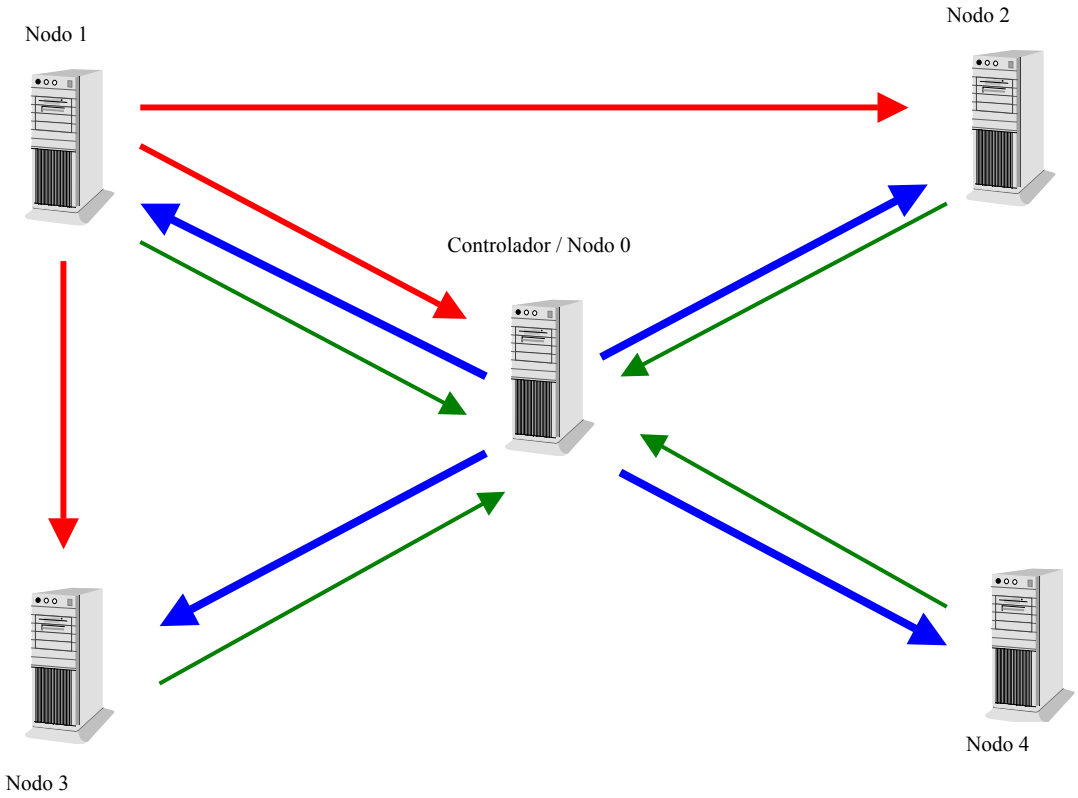
En un primer momento, se implementó un algoritmo genético utilizando una estructura de caminos. Esta representación tiene la característica de que se mantiene la factibilidad de los cromosomas luego de un cruzamiento o mutación de un gen del mismo. Esto es presentado como ventaja en [15], ya que se elimina automáticamente la necesidad de comprobar la factibilidad de un cromosoma luego de que fue creado por estos operadores.

Esta codificación presenta buenos resultados en problemas en que los caminos entre nodos terminales son mayoritariamente disjuntos. Sin embargo, cuando aumenta la cantidad de caminos y/o la cantidad de nodos terminales, se comienza a repetir material genético (una arista puede pertenecer a varios caminos, y si bien se quita de un gen, sigue presente en el resto). En casos extremos, las soluciones presentadas eran los grafos de entrada (Problema 3 de los casos de prueba del punto 6.2.2).

Debido a estos problemas, se reimplementó el algoritmo utilizando la estructura dada en 4.1.1. Cada cromosoma se inicializa con el grafo de entrada, exceptuando cero, una o dos aristas. Esto garantiza que no se pierde tiempo inicialmente generando cromosomas no factibles.

Esta implementación soluciona el problema mencionado arriba, a costo de tener que calcular la factibilidad de cada cromosoma cuando éste es generado. Este cálculo se basa en encontrar todos los caminos necesarios entre pares de nodos terminales.

4.2 Arquitectura del sistema



La topología propiamente dicha del sistema está dada por la comunicación entre procesos representada por las flechas rojas. Las flechas azules y verdes representan información administrativa entre los nodos del sistema y el proceso controlador.

El controlador es al mismo tiempo un nodo normal del sistema, autoenviándose los mensajes de control.

Cada nodo contiene una población ejecutando el mismo algoritmo genético, y las interacciones entre las distintas poblaciones están determinadas por las relaciones de migración y vecindad entre los mismos. Estas relaciones pueden variar de la siguiente manera:

- Tipo: Un sistema puede tener relaciones de migración, vecindad o ambas simultáneamente.

- Frecuencia: En el caso de que existan relaciones de migración, se puede indicar cada cuántas generaciones se realizan.
  En el caso de relaciones de vecindad, se puede especificar el porcentaje de cromosomas que se tomará de las poblaciones "vecinas".
- Cantidad: En todos los tipos de relación, se puede especificar la cantidad de cromosomas que participan de la misma.

Funciones del controlador:

El controlador mantiene información administrativa, además de correr el algoritmo genético. Esta información comprende:

- La mejor solución encontrada hasta el momento, en forma de cromosoma.
- La cantidad de generaciones que el proceso controlador ha procesado.
- El estado de todas las poblaciones, con respecto a la convergencia de las mismas a una solución.

Basándose en esta información, el controlador decide cuándo terminar todo el sistema, enviando un mensaje de control a cada población, e imprimiendo la mejor solución encontrada hasta el momento. La decisión puede ser tomada, en forma configurable por cualquiera de los siguientes criterios:

- Cantidad de generaciones procesadas.
- Tiempo de corrida.
- Convergencia de todas las poblaciones a soluciones locales.

Asimismo, el controlador mantiene una bitácora del sistema, indicando el estado de cada población en cada generación de la misma, con la fitness de la mejor solución, el fitness promedio de la población y el número de generación.

Funciones de los nodos:

Cada nodo mantiene una población, y corre el algoritmo genético sobre la misma. Luego de generar cada población, hace los siguiente:

- Envía un reporte de su estado al controlador. A este reporte lo llamaremos UPDATE.
- Recibe cromosomas dirigidos a él por relaciones de vecindad o migración.
- Envía cromosomas a las poblaciones especificadas en la topología dada.

El proceso termina cuando recibe un mensaje de control de controlador.

4.3 Modelos de paralelismo implementados

Nos vamos a basar en dos tipos de relaciones entre poblaciones:

1. Vecindad.
2. Migración.

4.3.1 Migración

En este caso, cada población tiene prefijada en el archivo de configuración un conjunto de poblaciones al cual enviar cromosomas. Esta relación se puede representar como un grafo dirigido donde los nodos son los vértices del mismo.

La migración se realiza al finalizar un número determinado de generaciones desde la última migración, al contrario de lo que sucede en las relaciones de vecindad, en las que siempre hay una posibilidad de seleccionar un cromosoma remoto.

4.3.2 Vecindad

En este tipo de topología, los antecesores de cada cromosoma son elegidos de la población local o, con una probabilidad variable, de una población "vecina"

Definimos la relación de vecindad de la siguiente manera: Una población local es vecina de un conjunto de poblaciones remotas si existe una probabilidad no nula de selección de un cromosoma de una de estas poblaciones remotas.

Sea p_0 la probabilidad de elección de un cromosoma en la población local. Si definimos un conjunto $\{x_i / i = 1..n\}$ de poblaciones vecinas, al iniciar el sistema hay una probabilidad $p_i = \frac{(1-p_0)}{n}$ de elegir un cromosoma de una población remota x_i.

A medida de que el sistema evoluciona, las probabilidades p_i varían de acuerdo a la fitness general de la población x_i asociada, aumentando cuando ésta aumenta y disminuyendo cuando ésta disminuye.

Las probabilidades son actualizadas en función del fitness de los barrios del sistema. Los barrios emiten por broadcast su fitness cuando observan una mejoría de la misma. Cada barrio recolecta los broadcast que recibe y actualiza las entradas en su tabla de vecindad.

El valor de p_0 es único para todos los procesos, siendo especificado en un archivo de configuración.

4.4 Pseudocódigo

4.4.1 Programa gen2

```
Inicializar mecanismo paralelismo y disparar procesos.
Leer opciones del programa
Si soy el proceso controlador{
        Inicializar bitácora
        Inicializar mejorCromosoma.
}

Generar población pop con sizepop cromosomas.

while(true){
        escalar Fitness de población.
        Genero población vacía newpop

        Mientras (tamaño de newpop < sizepop y no termine por condiciones de parada){
            Elegir cromosoma1 de pop.
            Elegir cromosoma2 de pop.

            Realizo cruzamiento y mutación de cromosoma1 y cromosoma2

            Si cromosoma1 es factible
                    Insertar cromosoma1 en newpop.
            Si cromosoma2 es factible
                    Insertar cromosoma2 en newpop.

           Si soy el proceso controlador{
                    Mientras hayan updates de procesos
                        Escribir update en bitácora.
                        Si solucion recibida es mejor que mejorCromosoma
                                mejorCromosoma = solucion recibida.
                        Si el control de convergencia esta habilitado{
                                Actualizar datos de convergencia del sistema.

                    Si se cumple alguna condicion de parada{
                        Enviar señal de parada al resto de procesos.
                        Presentar solucion.
                        Terminar.

            Si no soy el proceso controlador
                    Terminar si recibo señal de parada

            Recibir updates de vecindad

        }


        Enviar update con el mejor cromosoma de newpop al proceso controlador

        Procesar migracion si esta habilitada.
        Enviar updates de vecindad si está habilitada

        Borrar pop
        pop = newpop.
}
```

4.4.2 Función “recibir updates de vecindad”

```
Mientras (haya tanda de origen O esperando ser procesada)
        Para cada vecino V en tanda recibida
                Cromosoma C = decodificarVecino(V)
                Insertar C en cache de vecindad de O
```

4.4.3 Función “Enviar updates de vecindad”

```
Cantidad a enviar = tamaño de la población * porcentaje de envio
For i = 0 to cantidad a enviar
        Cromosoma C = poblacion.tirarRuleta()
        Codificar e insertar cromosoma en tanda de envío.

Para cada barrio del cual somos vecinos
        Enviar tanda de cromosomas.
```

4.4.4 Función “Procesar migración”

```
Mientras (haya tanda de inmigrantes esperando ser procesada)
        Para cada inmigrante I en tanda recibida
                Cromosoma C = decodificarInmigrante(V)
                Insertarlo en newpop.

Cantidad a enviar = tamaño de población * porcentaje de migración
For i = 0 to cantidad a enviar
        Cromosoma C = poblacion.tirarRuleta()
        Codificar e insertar cromosoma en tanda a enviar.

Para cada destino de migracion
        Enviar tanda de cromosomas codificados
```

Es de notar que el envío y recepción de cromosomas se hace de forma *asincrónica*. La biblioteca de paralelismo utilizada permite el envío y recepción de mensajes en forma no bloqueante, lo que nos permite seguir procesando el algoritmo original sin necesidad de confirmar que los mensajes llegaron a destino, o quedarnos bloqueados escuchando por mensajes dirigidos hacia nuestro proceso.

Condiciones de parada

El programa puede parar por las siguientes condiciones:

- Número de generaciones.
  Parar el sistema cuando el proceso controlador excede el número de generaciones fijado en la configuración.
- Tiempo.
  Parar el sistema cuando el proceso controlador ha excedido el tiempo de corrida (en segundos) fijado por el usuario.
- Intervención de usuario.
  Si se presiona Control-C (break), se detiene el sistema.
- Convergencia.
  Decimos que un proceso ha convergido cuando la relación entre la fitness promedio de su población actual y la fitness promedio de su población anterior es menor que un valor dado por el usuario. Se considera que el sistema ha convergido cuando el 75% de sus procesos lo han hecho.

Cada una de estas condiciones puede estar habilitada o no, de acuerdo a las opciones especificadas por el usuario. Siempre que el proceso termina, se presenta la mejor solución encontrada hasta el momento.

Updates

Todos los procesos envían al proceso controlador un mensaje al final de cada generación, con la siguiente información:

- Número del proceso que lo envía.
- El mejor cromosoma encontrado en la generación actual.
- El promedio de fitness de la generación actual.

El proceso controlador, al recibir los updates, realiza las siguientes acciones:

- Actualiza la información de convergencia del sistema
- Escribe en la bitácora del sistema una línea con la siguiente información: <Tiempo> <Proceso> <Número de generación> <Fitness promedio> <Mejor Fitness>.
  Ejemplo:

  ```
  6.565162    1    21.3 43
  ```

  Esto indica de que a los 6.57 segundos de haber comenzado el sistema, el proceso número 2 ha terminado la generación número 1, la cual tiene una fitness promedio de 21.3 y su mejor cromosoma tiene fitness 43.

4.5 Generación de números aleatorios

Se implementaron generadores de números pseudoaleatorios para poder reproducir experimentos. Hemos probado varios métodos estudiados en Knuth [4.3], y finalmente utilizamos un algoritmo que utiliza dos secuencias de números aleatorios generadas por métodos congruenciales lineal.

La fórmula de un método congruencial lineal es:

```
X(0) = S                                    (1)
X(n+1) = ( A*X(n) + C ) mod M               (2)
```

Donde A, C y M son constantes que deben ser elegidas según cierta regla para que la secuencia sea buena. La fórmula describe la forma de obtener la secuencia a partir de un valor inicial (semilla S). La ecuación (2) la escribiremos como MCG(A,X(n),C,M). El número M se lo llama *periodo* dado que la secuencia comienza a repetirse luego de sortear M números.

El algoritmo usado por nosotros utiliza dos secuencias generadas por dos métodos congruenciales lineales X e Y con distintas constantes, uno de estos métodos (en este caso la secuencia X) asignan valores a un array V, el otro elige un índice de este array de donde tomar y modificar el contenido del mismo. A continuación se describe el algoritmo para obtener el siguiente número de la secuencia aleatoria:

```
Y(n+1) = MCG(A2, Y(n), C2, M2)          (3)
indice = LARGO_BUFFER * Y(n+1) / M2     (4)
resultado = V[indice]                   (5)
X(n+1) = MCG(A1, X(n), C1, M1)          (6)
V[indice] = X(n+1)                      (7)
```

En la ecuación (4) se sortea el índice en el array `V de tamaño LARGO_BUFFER`, en la ecuación (5) se extrae el `resultado`, en las ecuaciones (6) y (7) se actualiza el valor del array en la posición del índice. Knuth [16] menciona que el periodo de este generador es considerablemente grande si los periodos `M1` y `M2` son números primos entre si.

Para que cada proceso del sistema paralelo tenga un generador aleatorio con diferentes secuencias no correlativas, elegimos diferentes parámetros A1, A2, C1, C2, M1 y M2.

4.6 Cálculo de factibilidad

Para determinar si un cromosoma es factible, se genera el grafo determinado por el mismo, y se verifica de que el mismo cumpla las restricciones del GSP.

Para ello, primero se realiza el siguiente test heurístico simple de bajo costo:

```
Para todo i dentro de vértices terminales
     Para todo j > i dentro de vértices terminales.
          k = cantidad de caminos necesaria entre i y j.
          Si grado(i) < k o si grado(j) < k
               El grafo no es factible y salgo del test.
```

Esta heurística es claramente $O(|R|^2)$.

Si el grafo pasa el test anterior, se encuentran los caminos existentes entre cada par de vértices terminales mediante la aplicación del algoritmo de Ford-Fulkerson [17] al grafo, para cada par de vértices terminales.

Como esta aplicación del algoritmo utiliza una función de capacidad de flujo unitaria (por lo tanto integral) para cada arista, el algoritmo termina en un tiempo $O(|E| * r_{max})$, siendo E el conjunto de aristas, y r_{max} el máximo $r_{ij} \in R$ [18]

Para ello, se toma al grafo como una red, asignando al primer vértice el rol de fuente y al segundo el rol de pozo. Cada arista del grafo se considera de capacidad unitaria. Ya que el algoritmo funciona sobre grafos dirigidos, se toma cada arista como un par de aristas de sentidos opuestos.

Luego de estas consideraciones, el algoritmo encuentra un flujo máximo entre la fuente y el pozo, el cual, debido a que todas las aristas tienen capacidad unitaria, coincide con el máximo de caminos disjuntos entre los mismos.

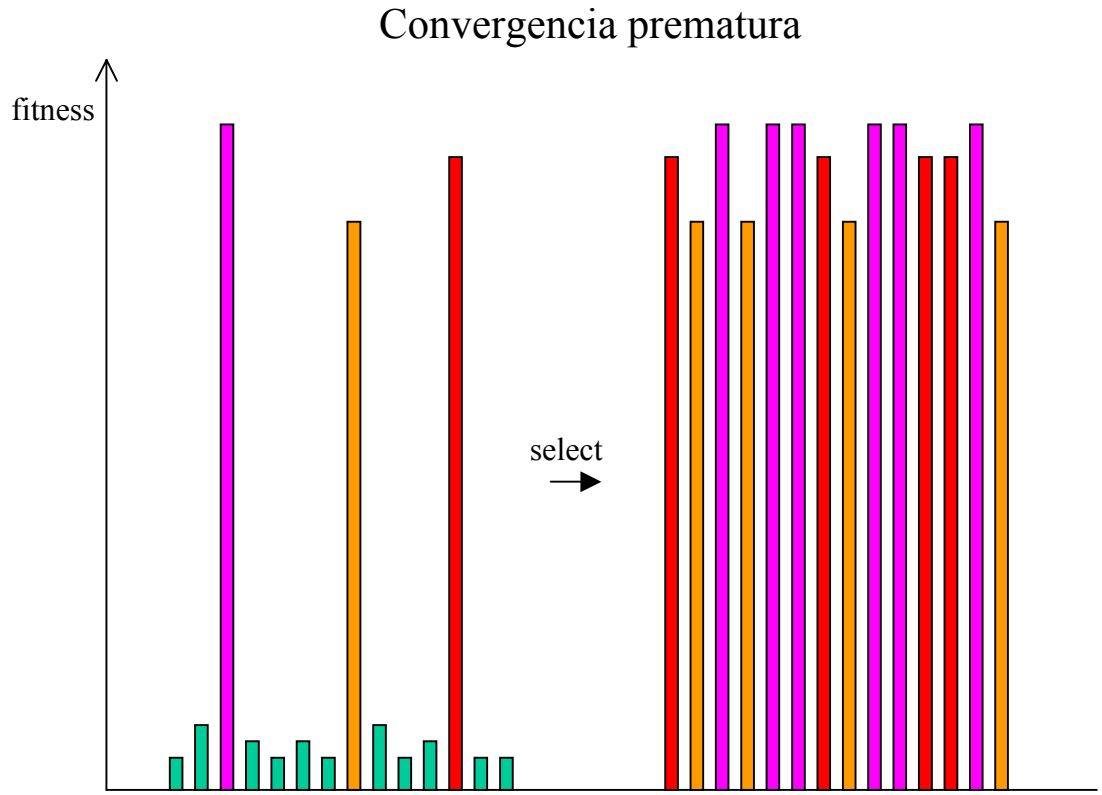
Como esto se hace para cada par de vértices terminales, el orden de esta segunda parte es de $O(|E| * r_{max} * |R|^2)$

Si la cantidad de caminos encontrada es menor que la necesaria, se toma al grafo como no factible y se sale del test.

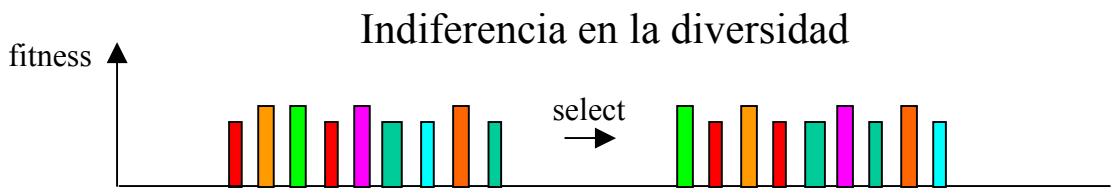
4.7 Fitness escalado

Comúnmente cuando las poblaciones poseen pocos individuos ocurren dos situaciones que hacen la selección un poco injusta si esta se basa en la función de fitness. Las situaciones son las siguientes:

a) Convergencia prematura.
La población posee pocos cromosomas extraordinarios y muchos cromosomas mediocres.
En éstas condiciones los pocos cromosomas buenos tomarán pocesión de la población en tan solo una generación lográndose una convergencia prematura.



b) Indiferencia en la diversidad.
La población posee una diversidad significativa, sin embargo todos los individuos tienen fitness parecidos. Consecuentemente la selección será indiferente frente a buenos y malos cromosomas.



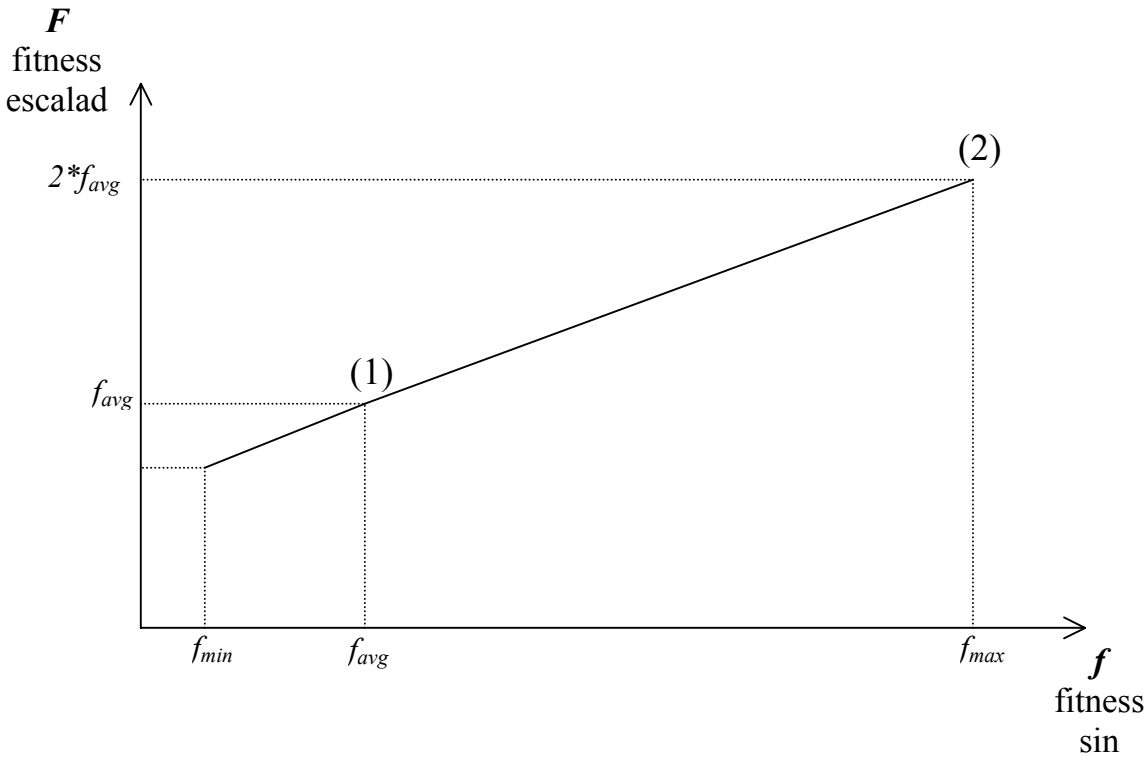
Para corregir estos comportamientos se implementó un escalamiento lineal del fitness. El mismo define una función lineal ***F*** que depende del promedio de fitness de la población f_{avg}, del máximo fitness de la población f_{max} y un factor de escalado ***C***. Las ecuaciones que rigen la función son:

(1) $F(f_{avg}) = f_{avg}$

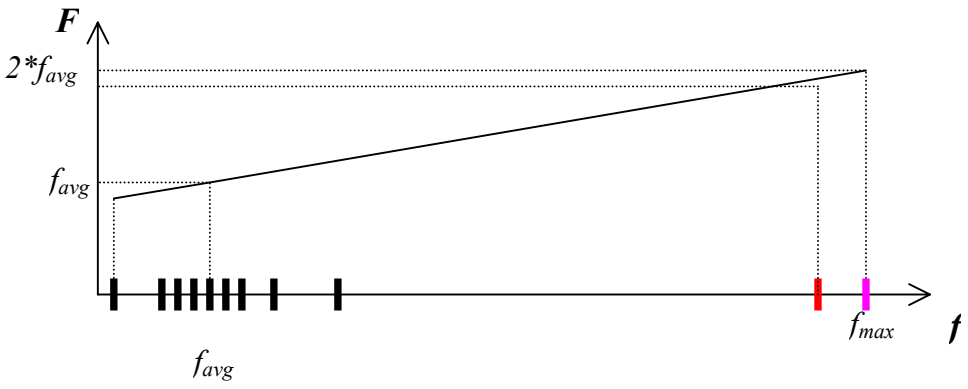
(2) $F(f_{max}) = C.f_{avg}$

La ecuación (1) asegura que los cromosomas medios tengan una copia en las nuevas generaciones. Mientras que la ecuación (2) regula el fitness de los cromosomas con fitness máximo. La constante ***C*** es el número de copias esperadas para el individuo con máximo fitness.

Una gráfica posible para ***F*** (con ***C*** = 2) es la siguiente:

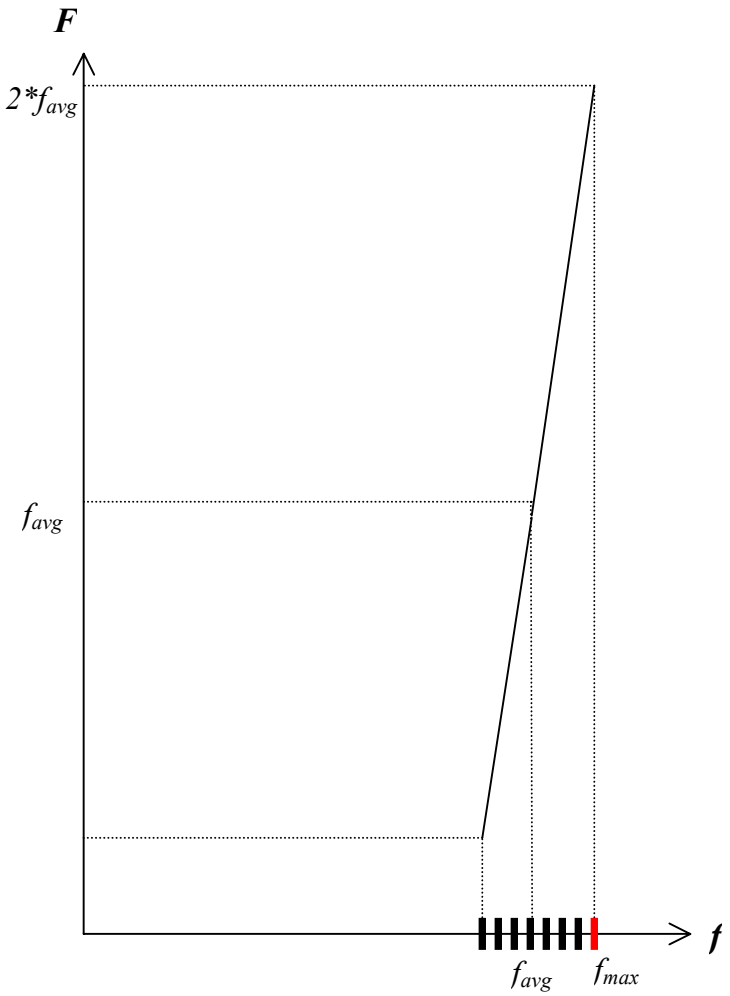


Veamos ahora gráficamente como el escalado resuelve el problema de convergencia prematura:



Los cromosomas extraordinariamente buenos tienen fitness escalado menores al fitness sin escalar.

En cuanto a la “indeferencia en la diverisidad”, el fitness escalado tiende a separar el fitness de los cromosomas de f_{avg}.



En resumen, se utiliza el fitness escalado en la selección para evitar estos comportamientos indeseados.

5 Consideraciones de implementación

5.1 MPI / MPICH

MPI significa Message Passing Interface. MPI es un standard que especifica una interfase de pasaje de mensajes entre procesos. La estandarización comenzó en abril de 1992 con el esfuerzo de mas 40 organizaciones mundiales en el área de la computación paralela. Se desarrollaron varias versiones del standard (1.x, 2.0). Desde abril de 1997 el fórum de MPI (MPIF) aceptó unánimemente la versión 2.0.

Para la codificación del algoritmo genético se utilizó una biblioteca llamada MPICH. El término 'MPICH' viene de 'MPI' y 'Chameleon', un sistema portable de comunicación de mensajes en el cual se basó la implementación inicial de MPICH. Este paquete es una implementación libre cubriendo totalmente el standard MPI 1.2, e implementando algunas de las funcionalidades de MPI 2.0. MPICH fue desarrollado (por el Laboratorio Nacional de Argonne de la Universidad de Chicago) en paralelo con el standard MPI para proveer puntos de discusión al MPIF.

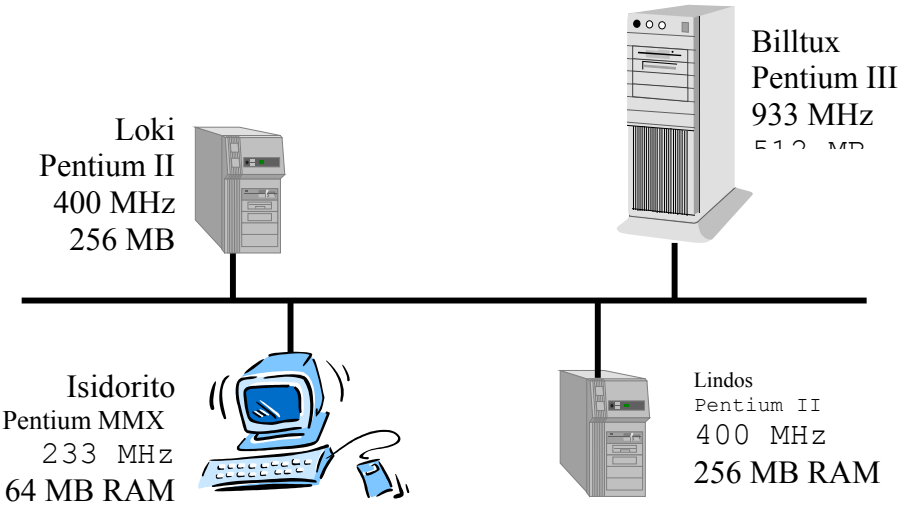
MPICH consiste en una biblioteca de rutinas MPI, compiladores (para código C, C++ y Fortran) y varios programas utilitarios.

Un proceso MPI es un programa C, C++ o Fortran que se comunica con otro proceso MPI por llamadas a las rutinas de la biblioteca. El usuario puede definir en número de procesos y en que máquinas desea ejecutar dichos procesos.

El lenguaje utilizado para programar el sistema fue ANSI C++, con el compilador G++ de GNU.

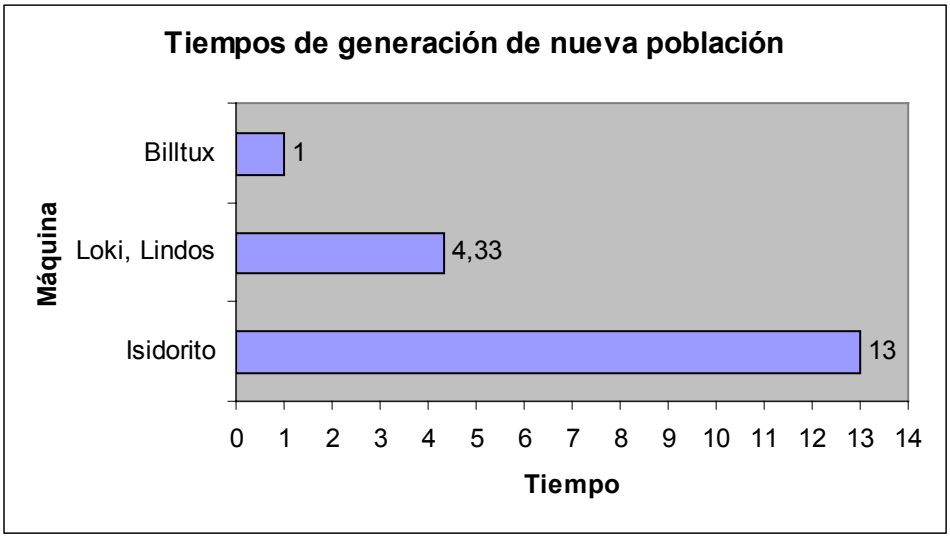
5.2 Sistemas de base utilizados – Hardware y software

Se utilizó una red IP sobre una LAN ethernet a 100 Mbps con cuatro computadoras heterogéneas con un solo procesador (Intel). En el dibujo a continuación se ilustra el hardware utilizado.



El sistema operativo instalado en Billtux, Loki y Lindos es SuSE Linux 6.3 y en Isidorito es Redhat Linux 7.1

5.3 Comparación de performance entre equipos



Estos resultados se obtuvieron midiendo el tiempo necesario para procesar una generación en cada uno de los equipos. O sea, si a Billtux le lleva 1 segundo procesar una población de determinado tamaño, a Isidorito le llevará 13 segundos, y 4.33 segundos a Loki y Lindos.

5.4 Cambios en el diseño

Pudimos observar algunos problemas en nuestro diseño inicial del algoritmo paralelo cuando comenzamos las pruebas de elección de parámetros. Básicamente los equipos utilizados son muy dispares en cuanto a capacidad de procesamiento de una generación.

En el diseño inicial del algoritmo las comunicaciones (asincrónicas, no bloqueantes) entre procesos ocurre al final de cada generación y el algoritmo consume el mayor tiempo en la producción de una nueva generación.

Los problemas comienzan cuando un proceso que se ejecuta en una máquina rápida satura de mensajes a otro proceso que se ejecuta en una máquina lenta. Lo que ocurre es que el proceso lento deja de comunicarse con los demás (en particular deja de reportar al proceso 0 sus updates).

Se implementaron dos soluciones:

a) Poblaciones adaptativas:

La idea fue equiparar el tiempo insumido en la generación por parte del proceso mas lento con los demás procesos del sistema. La maquina más lenta tendría una población fija (dada por el archivo de configuración). Máquinas rápidas tendrían poblaciones más grandes de tal manera que consumieran el mismo tiempo que la maquina más lenta.

Por ejemplo si el tiempo insumido por Isidorito en producir una nueva generación es 13 veces mas que en Billtux, entonces aumentamos a 13 el tamaño de la población de Billtux.

b) Poblaciones sincronizadas:

Simplemente cada proceso hace su generación y espera a que todos los demás terminen la misma generación. Aquí subestimamos el poder de las máquinas rápidas dado que los procesos en las maquinas rápidas tendrán que esperar por los lentos. Este es un modelo “justo” si se tuviera un cluster de equipos de iguales características.

Los resultados de este taller se basaron en la solución b) a pesar de tener un cluster de equipos heterogéneos.

6 Validación del sistema

6.1 Introducción

Para validar el sistema construido, se utilizó el mismo para encontrar soluciones a grafos de los cuales existe una solución conocida, ya sea exacta una aproximación.

Para ello, se utilizaron los grafos de prueba publicados en [6] lo que posibilita la comparación de las soluciones obtenidas con las presentadas en el trabajo citado.

Se utilizará la siguiente notación:

Los nodos rayados son terminales, los nodos en negro son de tipo Steiner. Los costos están indicados cerca de las aristas.

$R = (r_{ij})$ es la matriz restricciones. r_{ij} es el número de caminos (disjuntos respecto aristas) entre los terminales i, j. Cuando las mismas no son homogéneas, se da una matriz mostrando las restricciones entre cada par de vértices terminales.

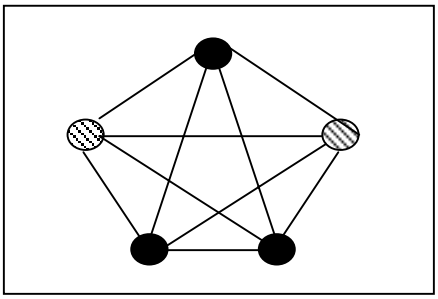
Los parámetros elegidos en todos los casos fueron los siguientes:

- Número de procesos: 1.
- Se ignoran los parámetros para migración y de barrios.
- Condición de parada: Maximo número de generaciones = 100
- Número de individuos en la población: 30
- Probabilidad de mutación: 0.00333
- Probabilidad de cruzamiento: 0.95
- Factor escalado: 2

6.2 Casos de prueba

Las soluciones publicadas fueron extraídas de [6].

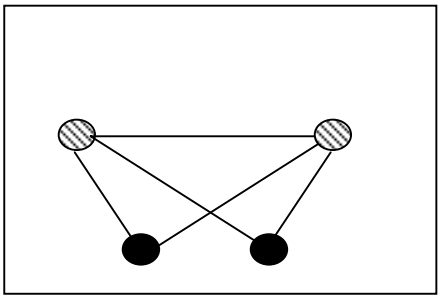
Problema 1



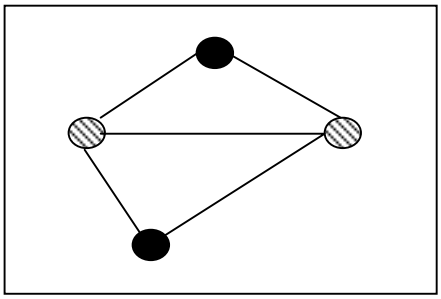
Todas las aristas tienen costo 1.

$$R = \begin{pmatrix} 0 & 3 \\ 3 & 0 \end{pmatrix}$$

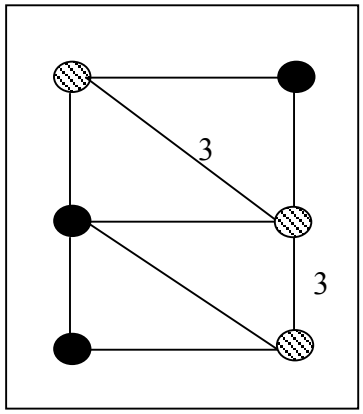
Solución encontrada por el algoritmo, costo 5.

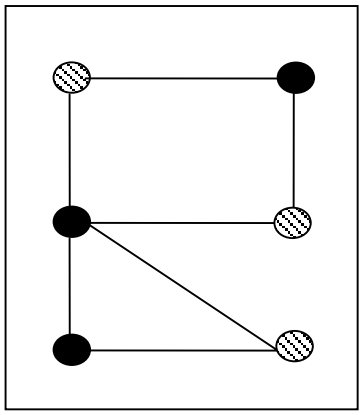


Solución publicada, costo 5



Problema 2



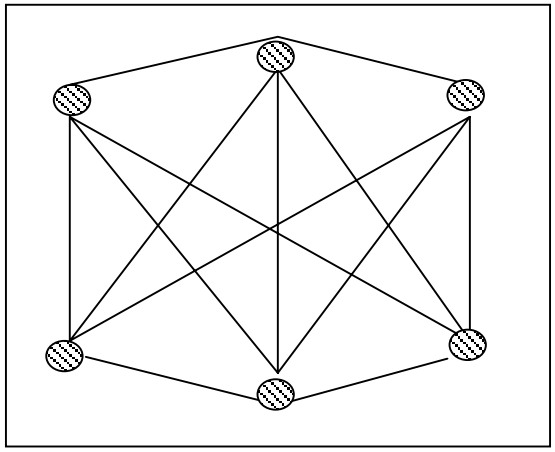


Solución encontrada por el algoritmo. Coincide con la solución publicada.

Las demás aristas tienen costo 1.

$$R = \begin{pmatrix} 0 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{pmatrix}$$

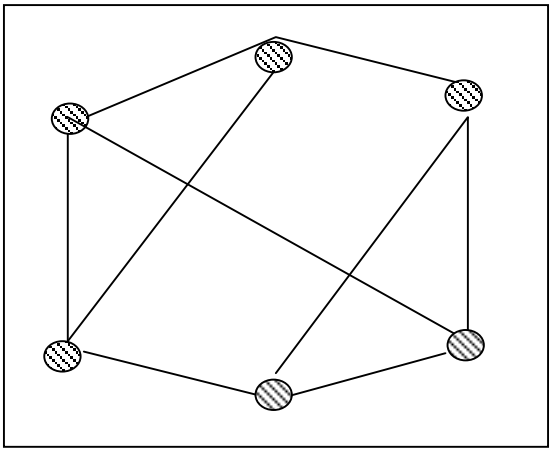
Problema 3



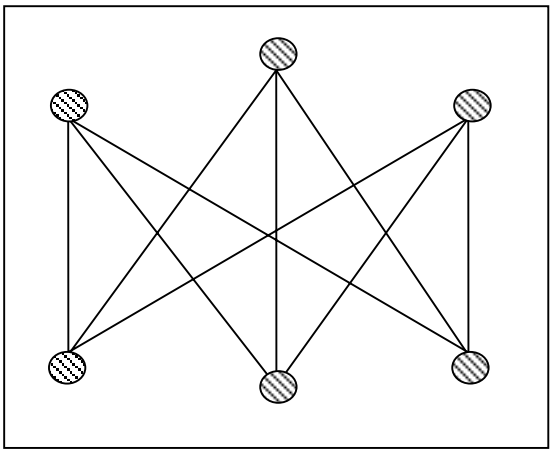
Todas las aristas tienen costo 1

$$R = \begin{pmatrix} 0 & 3 & 3 & 3 & 3 & 3 \\ 3 & 0 & 3 & 3 & 3 & 3 \\ 3 & 3 & 0 & 3 & 3 & 3 \\ 3 & 3 & 3 & 0 & 3 & 3 \\ 3 & 3 & 3 & 3 & 0 & 3 \\ 3 & 3 & 3 & 3 & 3 & 0 \end{pmatrix}$$

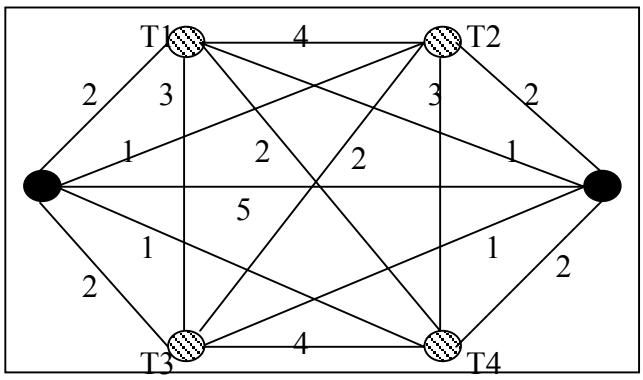
Solución encontrada por el algoritmo, costo 9.



Solución publicada, costo 9.

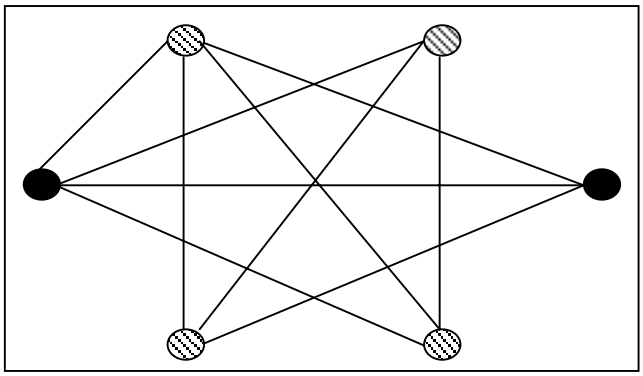


Problema 4

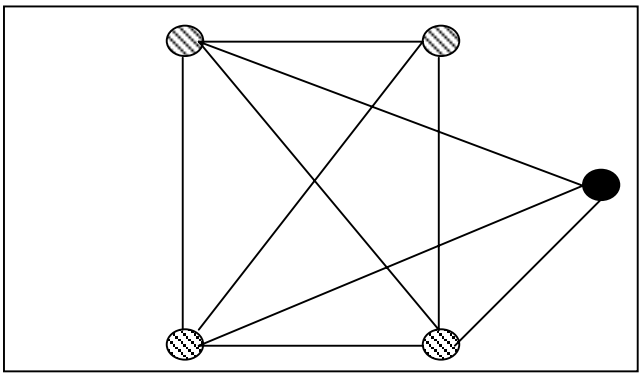


$$R = \begin{pmatrix} 0 & 2 & 3 & 4 \\ 2 & 0 & 2 & 3 \\ 3 & 2 & 0 & 2 \\ 4 & 3 & 2 & 0 \end{pmatrix}$$

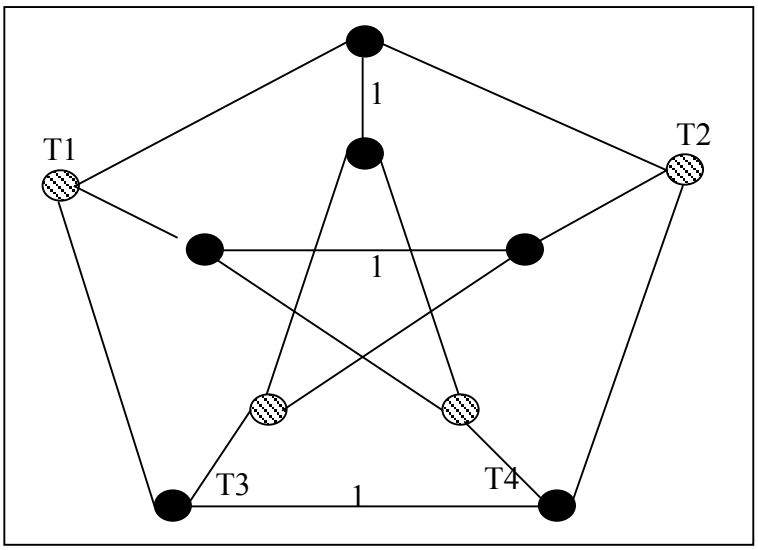
Solución encontrada por el algoritmo, costo 18.



Solución publicada, costo 22.



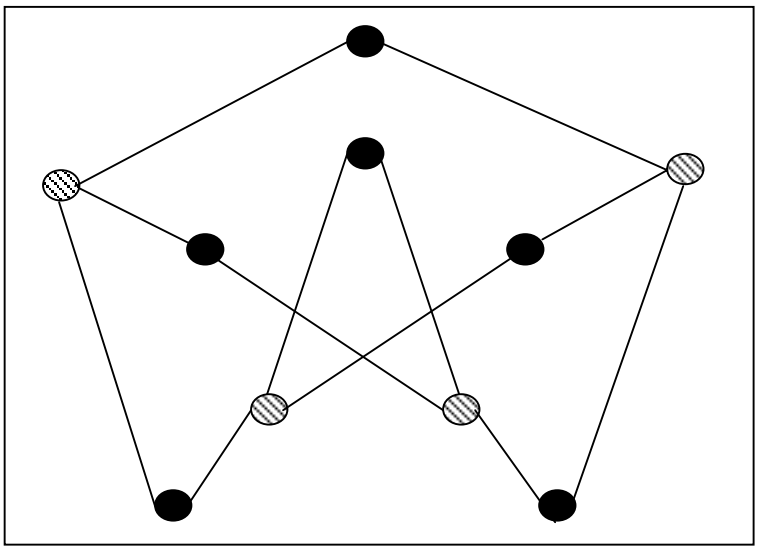
Problema 5



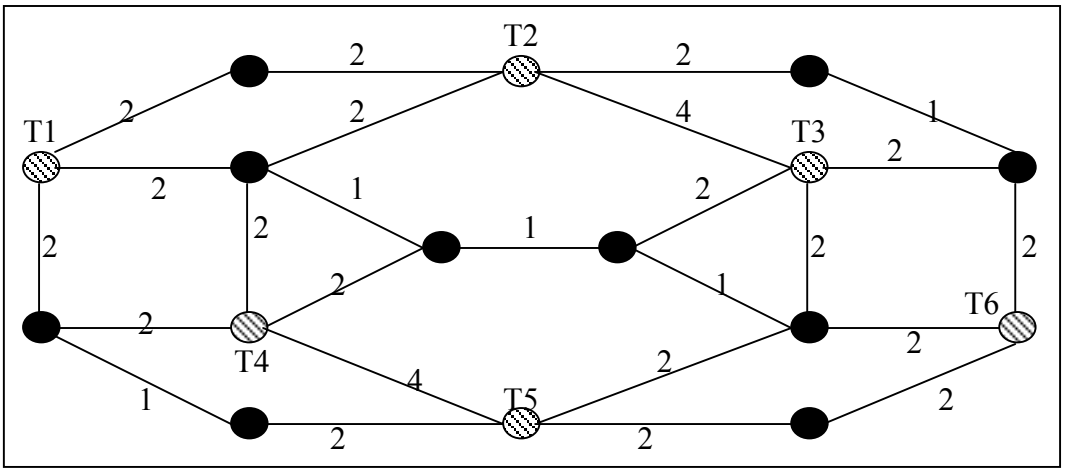
$$R = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 2 & 3 \\ 2 & 2 & 0 & 3 \\ 3 & 3 & 3 & 0 \end{pmatrix}$$

Las demás aristas tienen costo 2.

Solución encontrada por el algoritmo. Coincide con la solución publicada.

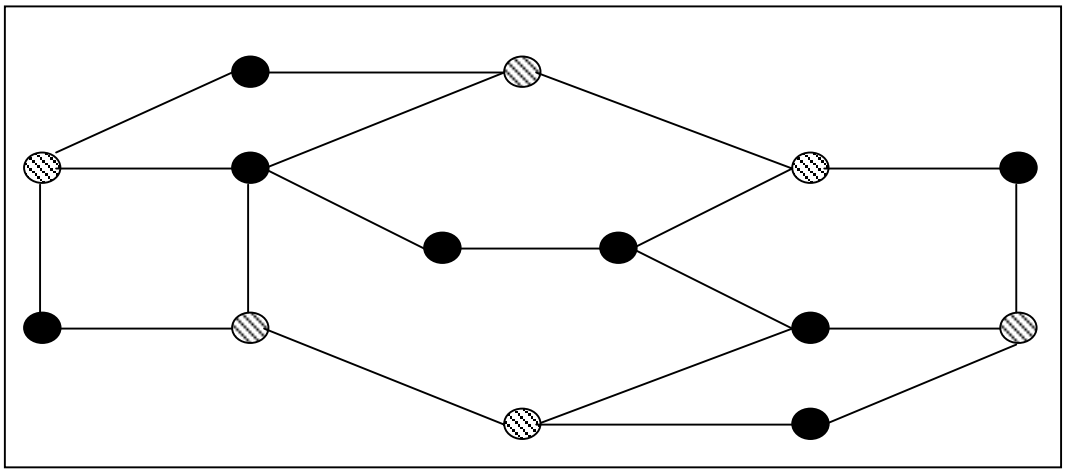


Problema 6

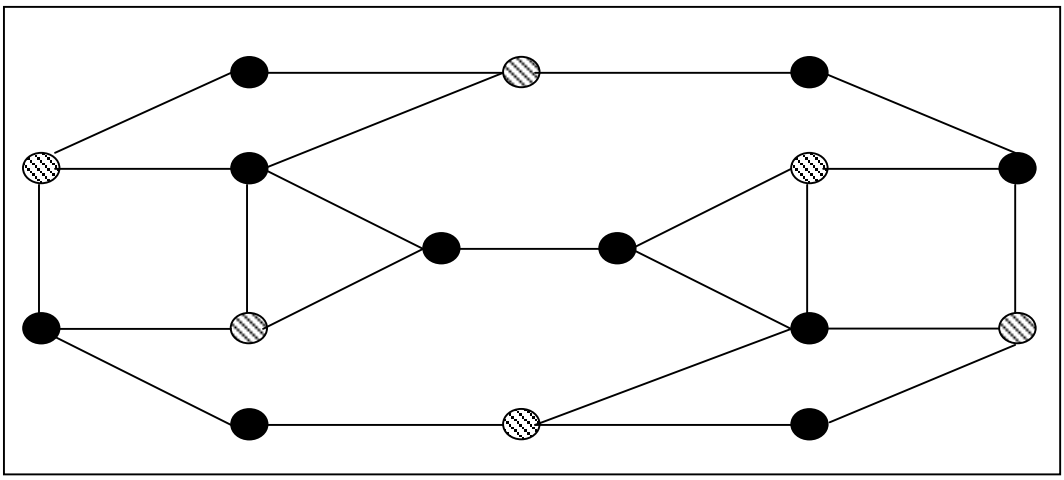


$$R = \begin{pmatrix} 0 & 2 & 3 & 2 & 3 & 2 \\ 2 & 0 & 2 & 3 & 2 & 3 \\ 3 & 2 & 0 & 2 & 3 & 2 \\ 2 & 3 & 2 & 0 & 2 & 3 \\ 3 & 2 & 3 & 2 & 0 & 2 \\ 2 & 3 & 2 & 3 & 2 & 0 \end{pmatrix}$$

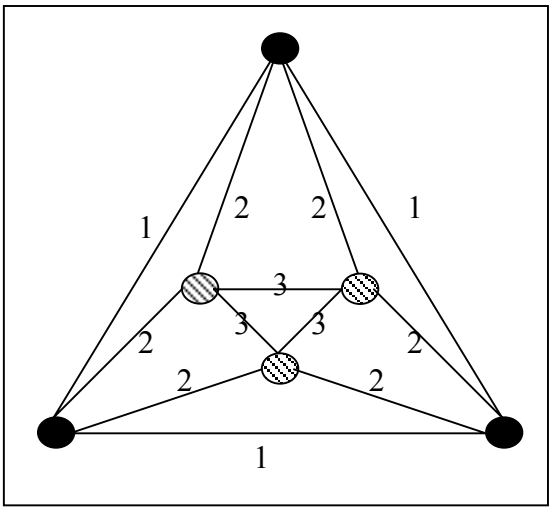
Solución encontrada por el algoritmo, costo 39.



Solución publicada, costo 41.

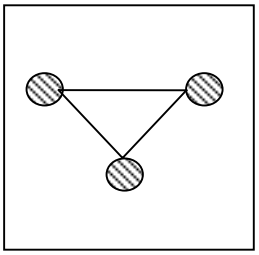


Problema 7



$$R = \begin{pmatrix} 0 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 0 \end{pmatrix}$$

Solución encontrada por el algoritmo. Coincide con la solución publicada.



6.3 Resultados

Las soluciones encontradas por el algoritmo coincidieron en costo con las soluciones publicadas en [6] para los problemas 1, 2, 3, 5 y 7. Para los problemas 4 y 6 se encontraron mejores soluciones que las publicadas.

7 ¿Vale la pena paralelizar?

7.1 Introducción

En el capítulo anterior se dan ejemplos en los que la heurística encontrada resuelve adecuadamente el GSNP. Una vez pasada la primer etapa, de mostrar que un algoritmo genético puede resolver adecuadamente el problema planteado, la pregunta que se plantea inmediatamente es ¿Cómo hacerlo más rápidamente?.

Asumiendo que no hay posibilidades de obtener hardware más rápido, la forma que nos queda de acelerar la obtención de soluciones es la de distribuir su procesamiento en distintos procesadores.

Esto se puede hacer de distintas maneras; desde una más simple que consiste en correr el mismo algoritmo en paralelo en distintos procesadores y tomar al cabo de determinado tiempo la mejor solución encontrada hasta formás más complicadas implicando comunicaciones entre los distintos procesos.

Cada una de estas alternativas puede dar soluciones distintas al GSNP (dado que cada una tiene como base un algoritmo genético, el cual, por su carácter de heurística no da soluciones demostrablemente finales), por lo que es de interés el comparar la calidad de los resultados obtenidos con algunas de ellas.

7.2 Experimento

7.2.1 Consideraciones previas

Para obtener cierto grado de generalización de los resultados se generaron cinco grafos con las siguientes características:

- 100 vértices
- 500 aristas
- 20 vértices terminales
- requerimientos de conexión entre cada par de terminales aleatorios uniformes entre 0 y 4.

Usaremos la siguiente notación para describir estos grafos:

grafo(100, 500, 20, 0..4)

En las pruebas de validación se vio que el algoritmo resolvía muy rápidamente grafos pequeños, del tipo utilizado en las mismas. Asimismo, la solución encontrada era la misma en todos los casos.

Para poder analizar mejor el efecto de cada alternativa utilizada se decidió utilizar grafos más grandes. Dado que el algoritmo genético es $O(|R|^2 * r_{max} * |E|^2)$ para un tamaño de población fija, y dada la gran cantidad de casos de prueba a correr, se eligió el tamaño de grafo especificado arriba como un buen término medio entre ambos factores (suficiente complejidad como para que haya variabilidad en las soluciones y tamaño manejable para correr múltiples casos de prueba).

Los grafos fueron generados con el mismo algoritmo genético utilizados para resolverlos, cambiando la función de fitness. Lo detalles se pueden encontrar en el apéndice correspondiente.

7.2.2 Experimento

1. Se resolvió un problema de Steiner generalizado en grafos en una única máquina con cinco grafos de prueba con las mismas características. Se corrió el algoritmo con una población de tamaño M = 120 hasta un límite de generaciones arbitrario (300), registrando el promedio de fitness de las soluciones obtenidas.

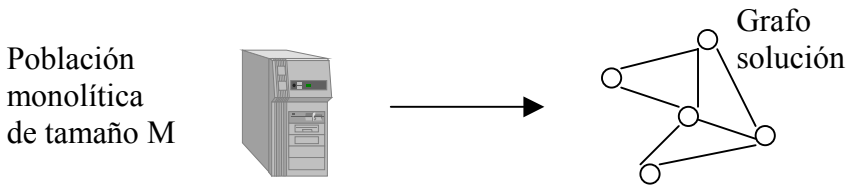


Figura 7.1

2. Se dividió la población entre cuatro y se asignó cada subpoblación a un ordenador distinto. Se corrió el algoritmo genético en cada ordenador y se registraron soluciones locales a medida que se iban obteniendo tomando la mejor de ellas como solución final del problema.

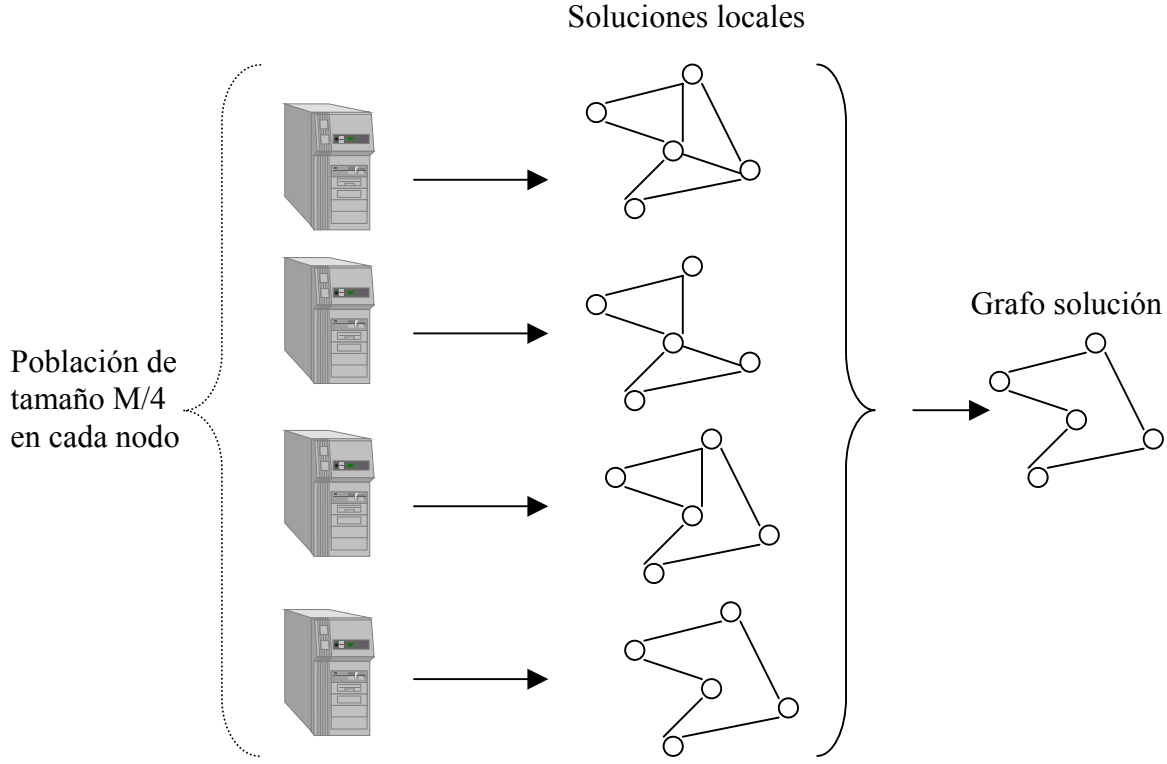


Figura 7.2

3. Repetimos el experimento 2 con el mismo número de máquinas y el mismo tamaño de población local (M/4), pero permitiendo intercambio de material genético entre las poblaciones.

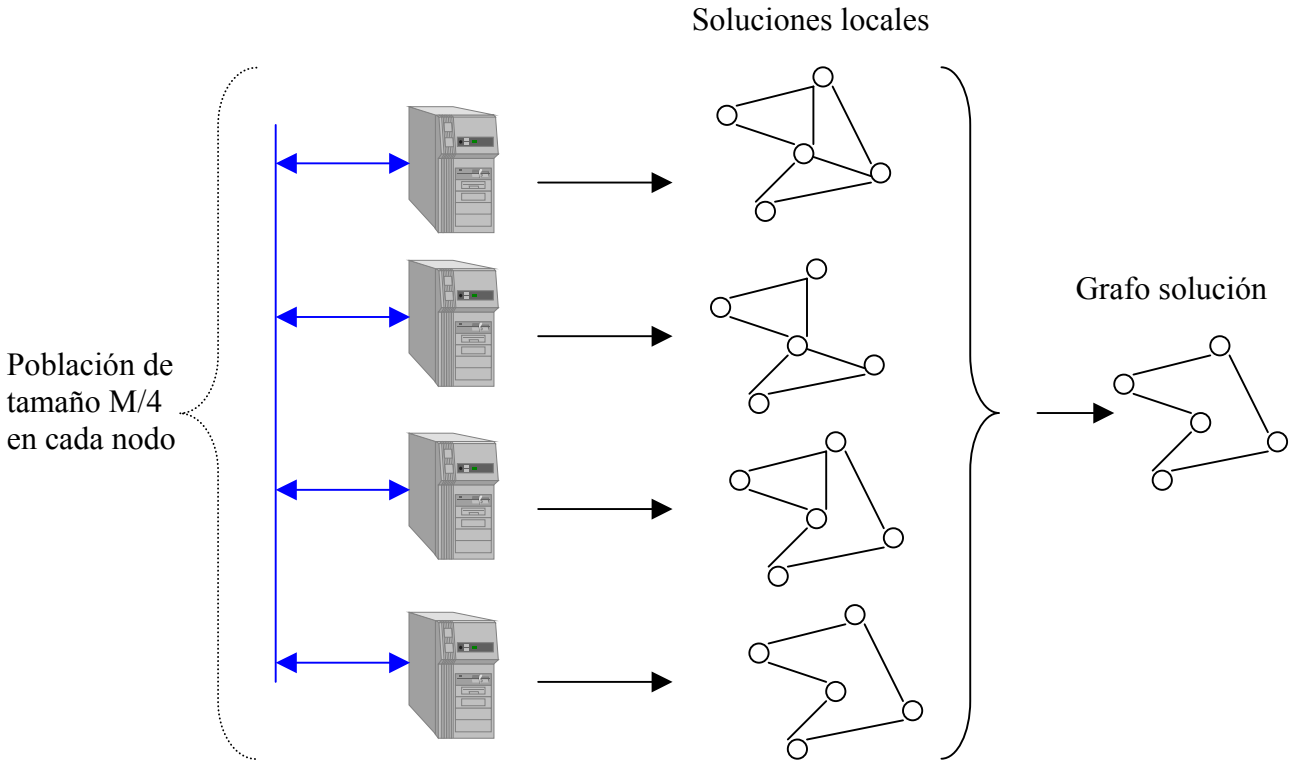


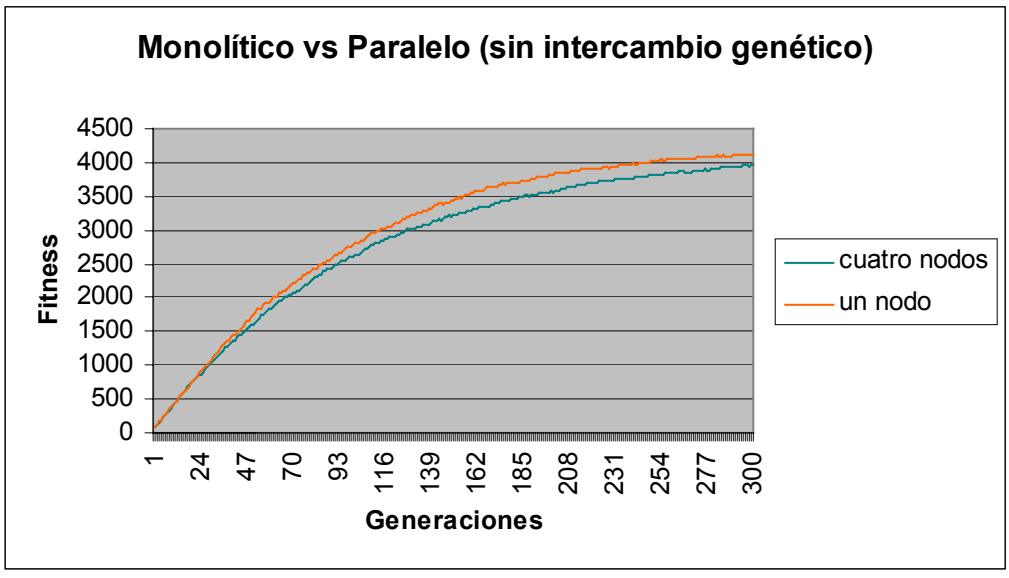
Figura 7.2

7.3 Resultados

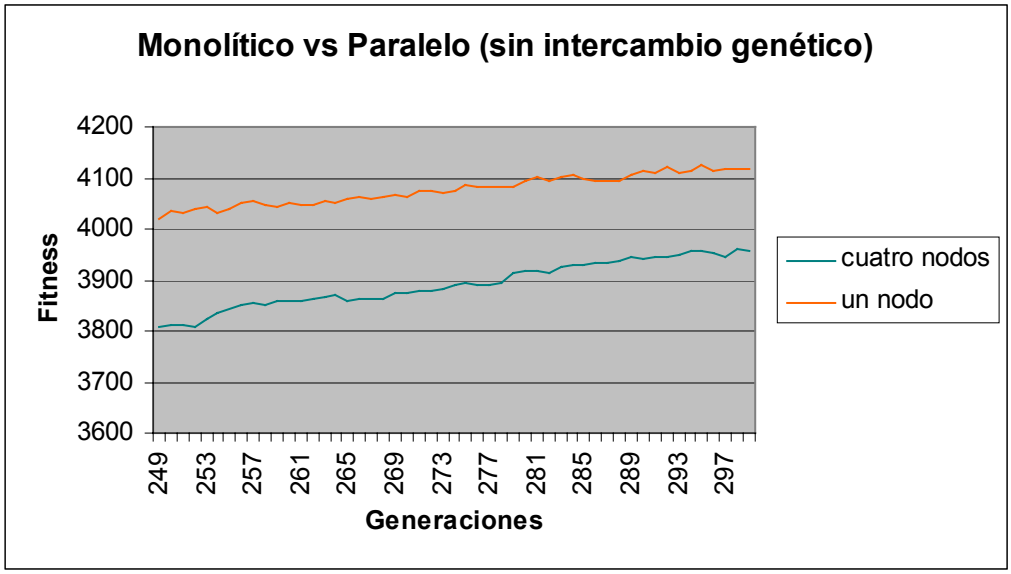
Las corridas se hicieron con los siguientes parámetros:

Modelo	Monolítico	Paralelo
Número de máquinas	1	4
Tamaño de la población (***SIZEPOP***)	120	30
Parámetros con iguales valores en ambos modelos		
Grafos (vértices, aristas, terminales, restric)	cinco grafo(100,500,20,0..4)	
Condición de parada	300 generaciones	
Probabilidad de mutación (***PMUT***)	0.00333	
Probabilidad de cruzamiento (***PCROSS***)	0.95	
Factor de escalado del fitness (***C***)	2	
Topología migración	ignorado	
Porcentaje de migración	ignorado	
Periodo o Epoca de migración	ignorado	
Topología vecindad	ignorado	
Porcentaje de vecindad	ignorado	
Probabilidad de elección local	ignorado	

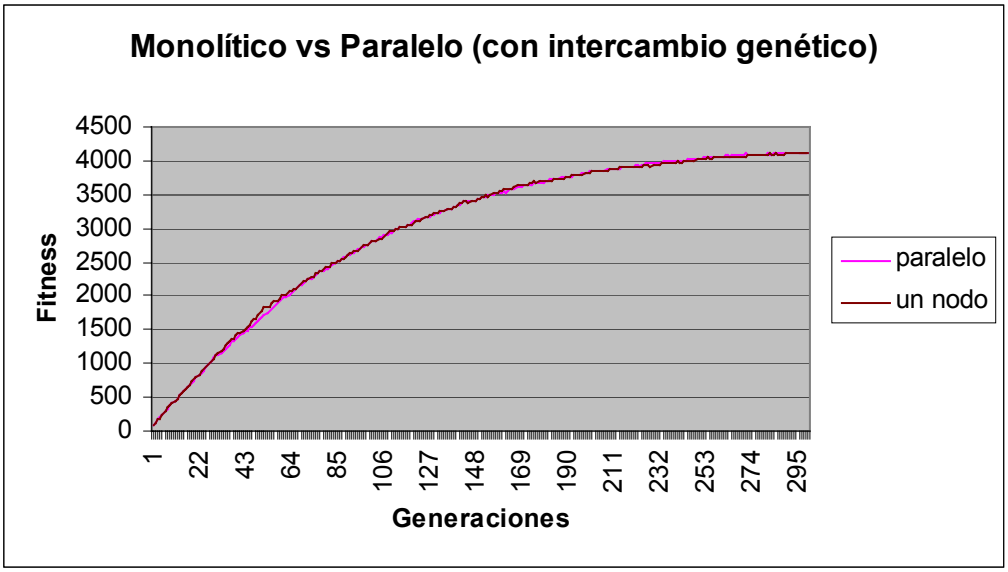
Las siguientes gráficas permiten comparar los resultados de los experimentos anteriores en función del número de generaciones. Para obtener estas gráfica, para cada generación, se hizo un promedio de los máximos correspondientes a la generación.



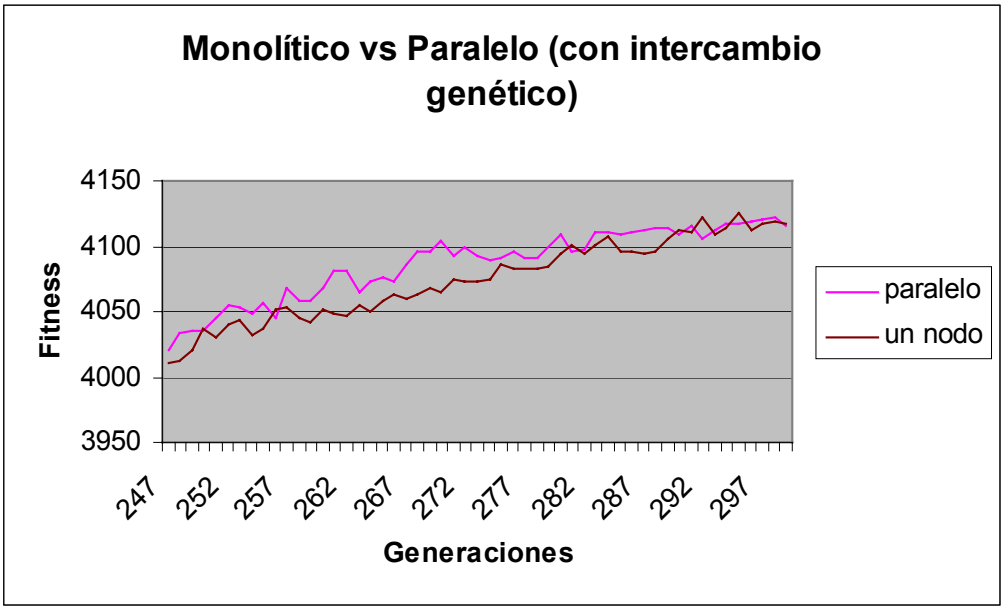
A continuación ampliamos las últimas 50 generaciones de la gráfica anterior.



Ahora compararemos el modelo monolítico con el paralelo con intercambio de cromosomas.



A continuación ampliamos las últimas 50 generaciones de la gráfica anterior.



7.4 Observaciones

a) El modelo monolítico devuelve mejores resultados que el paralelo sin intercambio genético.
b) Los resultados del modelo paralelo con intercambio de material genético son comparables con los obtenidos del modelo monolítico. Observamos que la máquina que corre el modelo monolítico tiene una población cuatro veces más grande que las máquinas que corren en el modelo paralelo.
c) Dado que el tiempo de ejecución del algoritmo genético es linealmente proporcional al tamaño de la población, el modelo monolítico termina en un tiempo cuatro veces superior al de los modelos paralelos.

7.5 Conclusiones

Para obtener buenos resultados paralelizando este problema no basta con la solución simple de partir la población en distintos procesadores. Es necesario investigar un modelo de comunicación entre las poblaciones. Este estudio lo haremos en los siguientes capítulos.

8 Estudio del modelo de migración

8.1 Introducción

En este capítulo aplicamos el modelo de migración en demes para resolver el problema de Steiner investigando la sensibilidad de los resultados en función de los parámetros que determina el modelo.

Estos parámetros son:

- *Período* o *Época*: Indica cada cuantas generaciones se emigran individuos
- *Porcentaje de Migración*: Es el porcentaje de individuos de la población local que emigra
- *Topología*: Se expresa como un grafo dirigido cuyos nodos representan poblaciones. Una arista de la población P1 hacia la población P2 significa que se migrarán individuos de P1 hacia P2.

Los valores considerados en estas pruebas fueron:

Número de máquinas	4
Porcentaje de migración	1,5,10,20,30,40,50
Período	1,2,4,6,8,10
Topología	grafo completo, anillo, anillo+1
Grafos (vértices, aristas, terminales, restric)	cinco grafo(100,500,20,0..4)
Condición de parada	300 generaciones
Tamaño de la población (***SIZEPOP***)	30
Probabilidad de mutación (***PMUT***)	0.00333
Probabilidad de cruzamiento (***PCROSS***)	0.95
Factor de escalado del fitness (***C***)	2

Topologías de migración

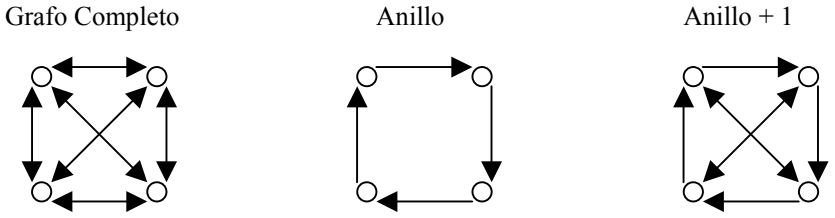


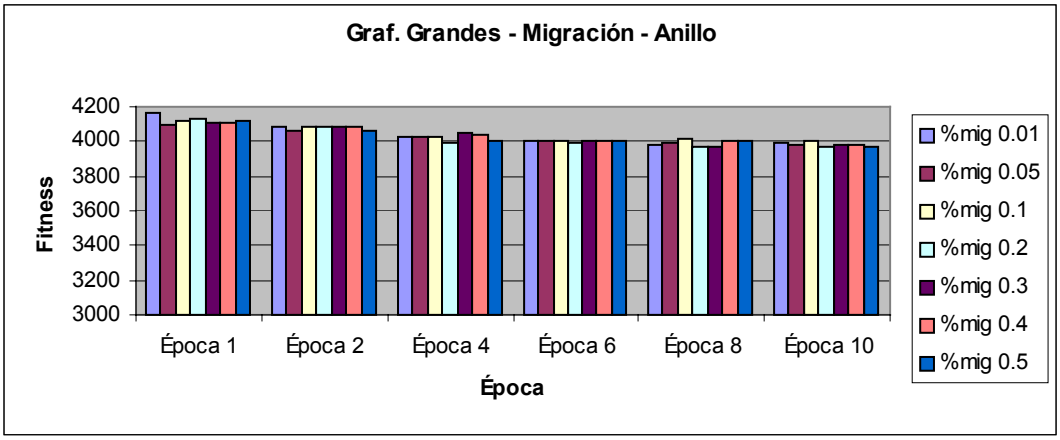
Figura 8.1

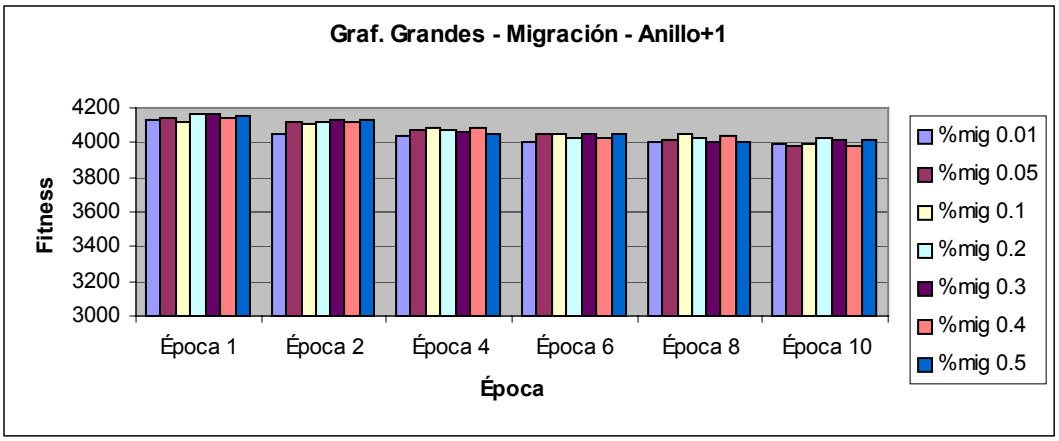
Se tomaron estos parámetros para abarcar una cantidad representativa de configuraciones del modelo, manteniendo la misma limitada por disponer de tiempo limitado de ejecución.

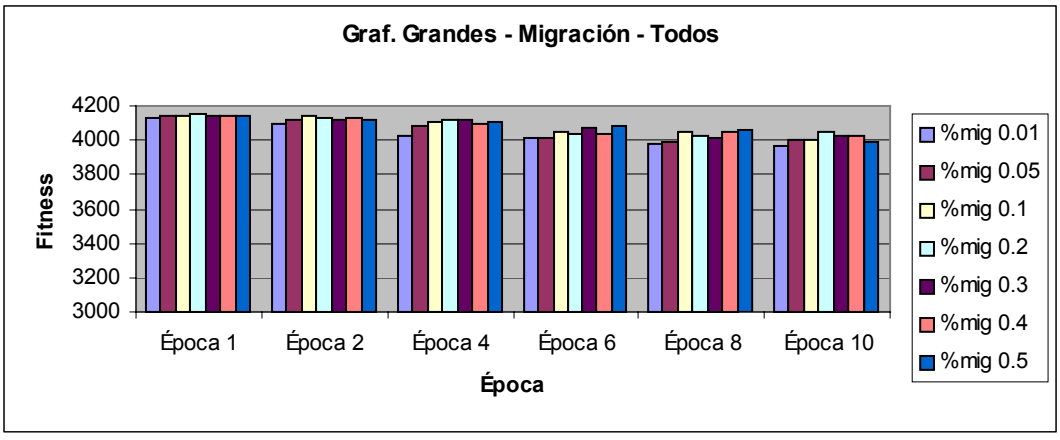
8.2 Fitness vs Época

8.2.1 Gráficas

A continuación se muestran gráficamente los resultados de las corridas en función de la época para cada topología.







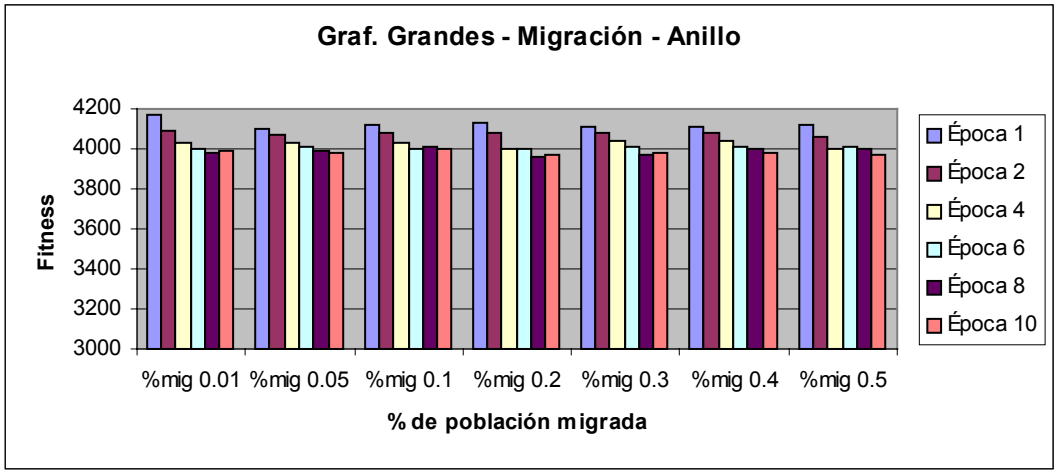
8.2.2 Observaciones

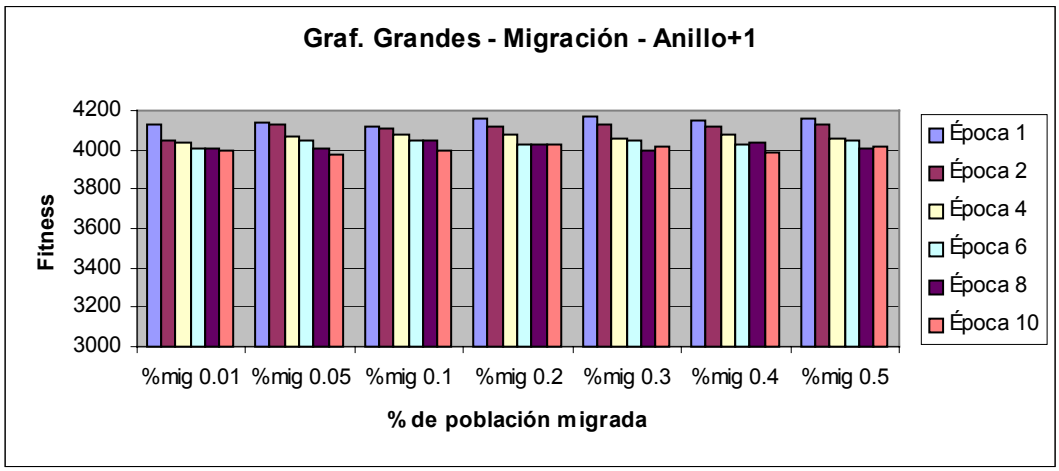
- No se aprecian diferencias significativas en los resultados finales según la topología utilizada. El máximo ocurre en la topología anillo+1.
- En todas las topologías los mejores resultados ocurren con un período de migración igual a uno.

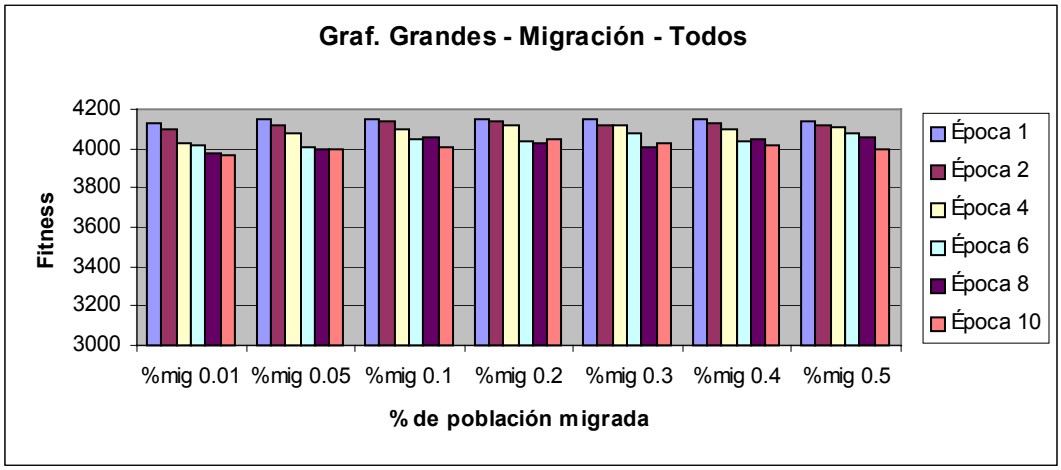
8.3 Fitness vs Porcentaje de migración

8.3.1 Gráficas

A continuación se muestran gráficamente los resultados de las corridas en función del porcentaje de migración.







8.3.2 Observaciones

- No se aprecian diferencias en los resultados finales según el porcentaje de migración utilizada.
- No se aprecian diferencias en los resultados al variar la topología.

8.4 Conclusiones

En función de cada parámetro considerado:

- *Topología:* No se aprecian diferencias significativas en los resultados de las pruebas. El máximo se obtuvo con la topología anillo+1.
- *Porcentaje de migración:* La cantidad de cromosomas migrados no afectó el resultado de las pruebas.
- *Época o Periodo:* Los mejores resultados se obtuvieron consistentemente con período uno. Los resultados empeoran monótonamente a medida que aumenta el período.

9 Estudio del modelo de vecindad

9.1 Introducción

En este capítulo aplicamos el modelo de vecindad para resolver el problema de Steiner investigando la sensibilidad de los resultados en función de los parámetros que determina el modelo.

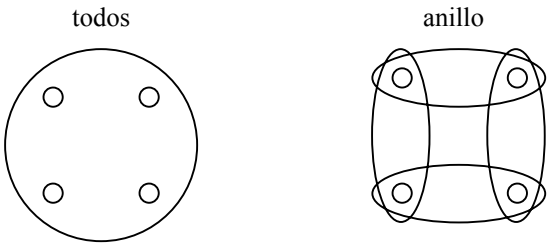
Estos parámetros son:

- *Probabilidad de elección local*: Es la probabilidad de seleccionar un individuo de la población local.
- *Porcentaje de vecindad*: Es el porcentaje de individuos de la población local que son transferidos a los barrios vecinos.
- *Topología*: Se define una relación de vecindad entre barrios de la siguiente manera: Dos barrios B1 y B2 están en la misma relación si el cruzamiento se puede dar entre individuos de B1 y B2.

Los valores considerados en estas pruebas fueron:

Número de máquinas	4
Porcentaje de vecindad	1,5,10,20,30,40,50
Probabilidad de elección local	99,95,90,80,60,50
Topología	todos, anillo
Grafos (vértices, aristas, terminales, restric)	cinco grafo(100,500,20,0..4)
Condición de parada	300 generaciones
Tamaño de la población (***SIZEPOP***)	30
Probabilidad de mutación (***PMUT***)	0.00333
Probabilidad de cruzamiento (***PCROSS***)	0.95
Factor de escalado del fitness (***C***)	2

Topologías de vecindad

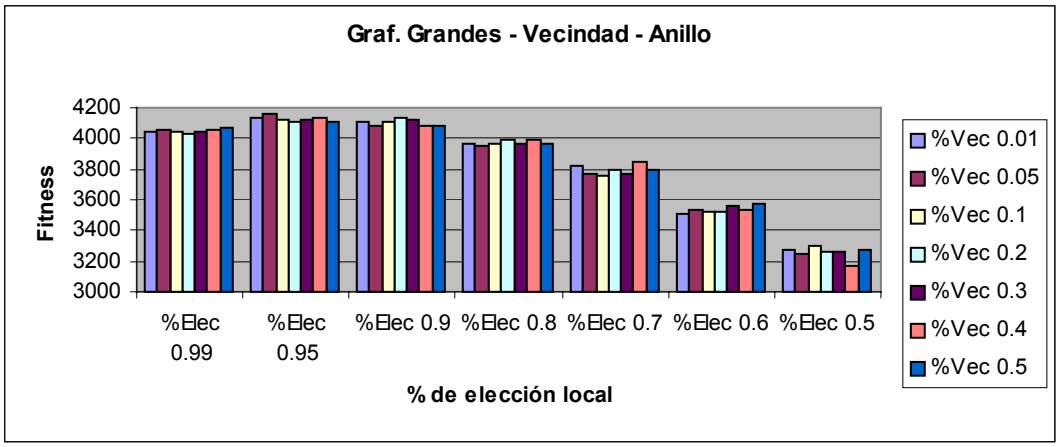


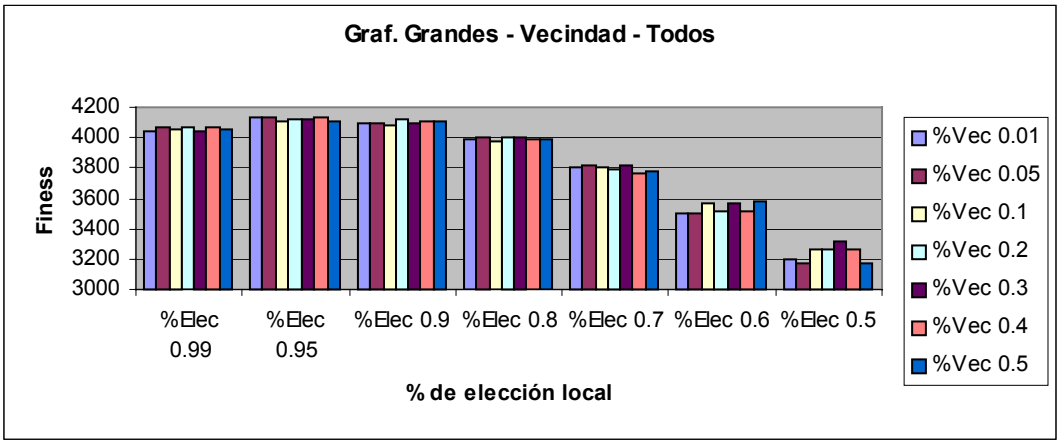
Se tomaron estos parámetros para abarcar una cantidad representativa de configuraciones del modelo, manteniendo la misma limitada por disponer de tiempo limitado de ejecución.

9.2 Fitness vs Porcentaje de elección local

9.2.1 Gráficas

A continuación se muestran gráficamente los resultados de las corridas en función del porcentaje de elección local.





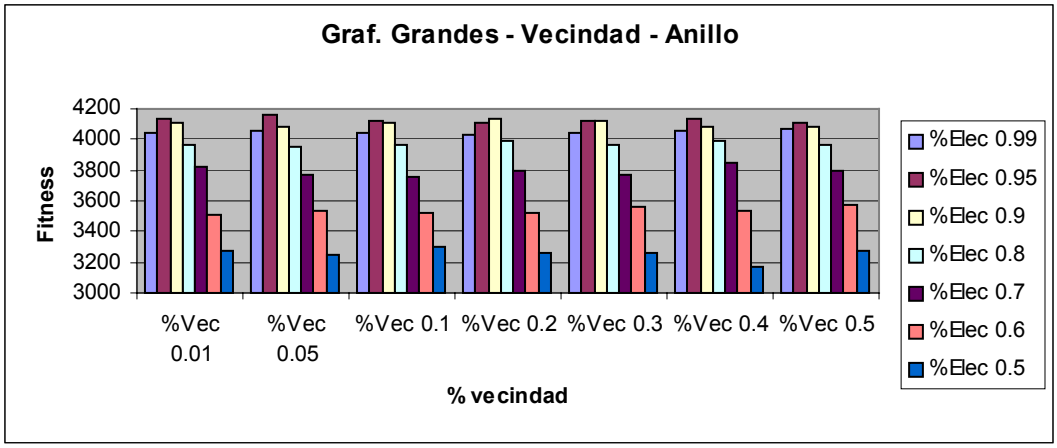
9.2.2 Observaciones

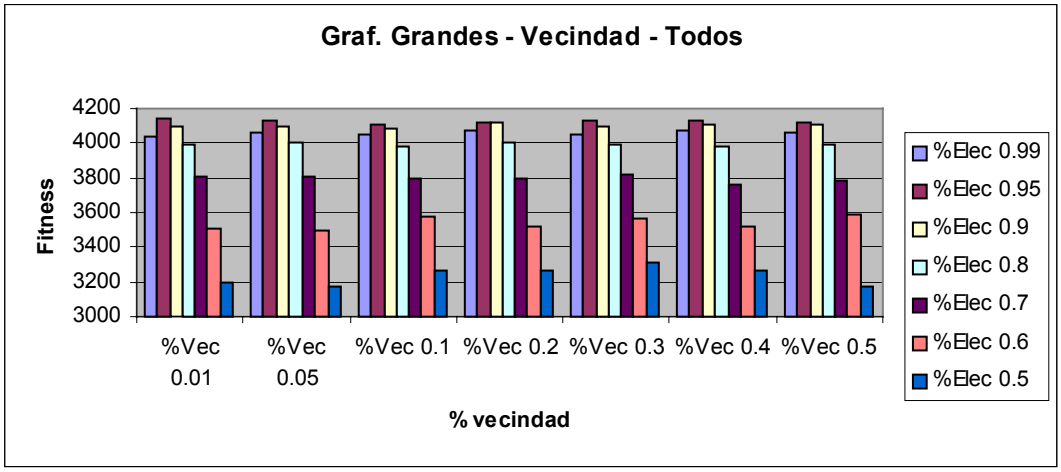
- No se aprecian diferencias significativas en los resultados al variar la topología.
- Los mejores resultados se observan al tomar porcentajes de elección local entre el 90% y el 95%.
- A medida que disminuye el porcentaje de elección local los resultados empeoran considerablemente.

9.3 Fitness vs Porcentaje de vecindad

9.3.1 Gráficas

A continuación se muestran gráficamente los resultados de las corridas en función del porcentaje de individuos de la población local que son transferidos a los barrios vecinos.





9.3.2 Observaciones

- No se aprecian diferencias en los resultados finales según el porcentaje de vecindad.
- No se aprecian diferencias en los resultados al variar la topología.

9.4 Conclusiones

En función de cada parámetro considerado:

- *Topología:* No se aprecian diferencias significativas en los resultados de las pruebas. El máximo se obtuvo con la topología anillo.
- *Porcentaje de vecindad:* La cantidad de cromosomas enviados a los barrios vecinos no afectó el resultado de las pruebas.
- *Porcentaje de elección local:* Los mejores resultados se obtuvieron consistentemente con valores entre 90 y 95%. Los resultados empeoran monótonamente a medida que disminuye el porcentaje de elección de local.

10 Comparación de modelos

10.1 Introducción

En este capítulo comparamos los modelos hasta ahora estudiados. Los mismos son:

- Migración
- Vecindad
- Monolítico

Las comparaciones se basaron en:

- la evolución del fitness para cada generación y
- el tiempo de procesamiento

Primeramente se compara el modelo de Migración con el modelo de Vecindad. Luego se compara Migración con el modelo Monolítico.

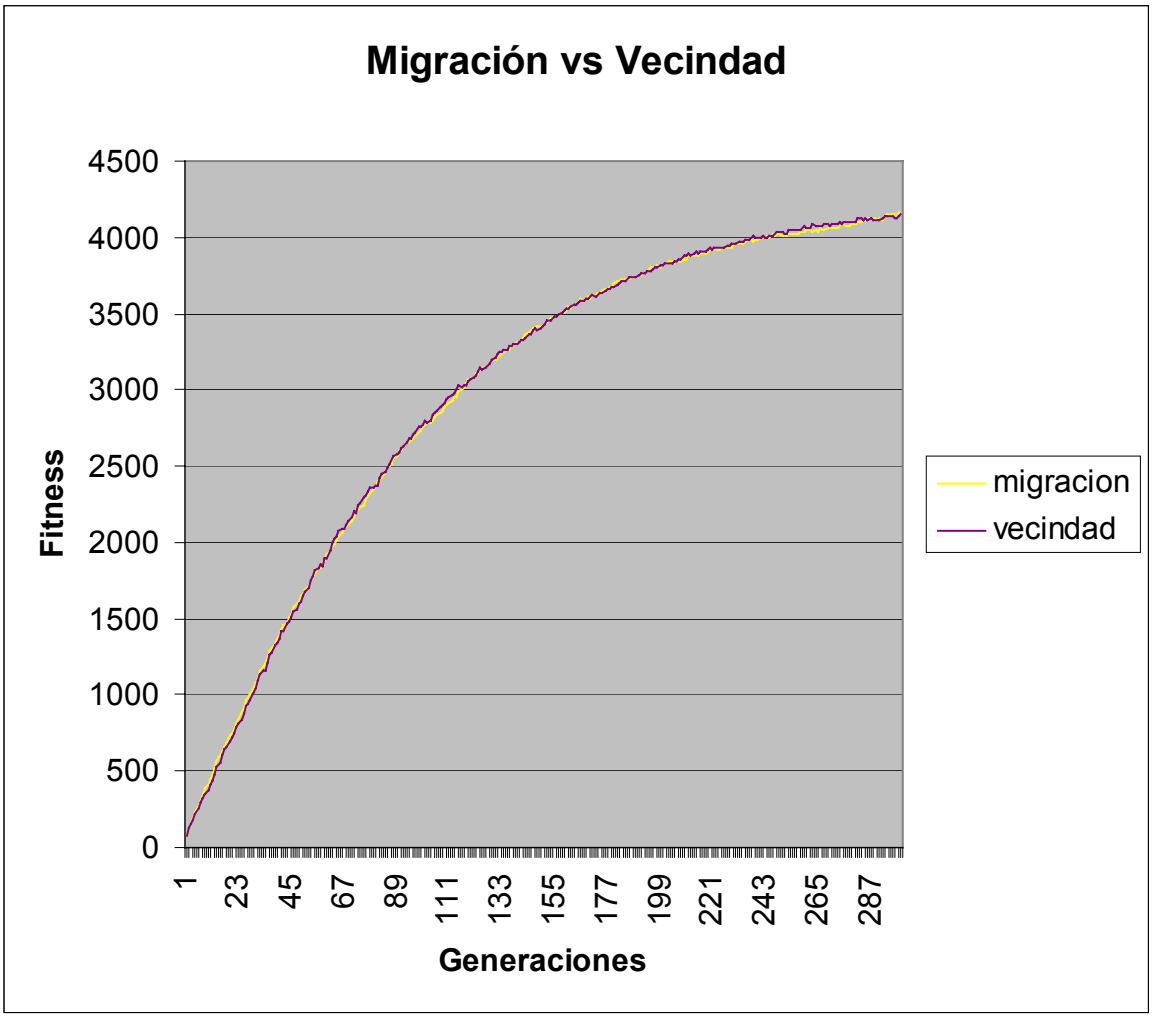
10.2 Migración vs Vecindad

Para poder comparar ambos modelos (Migración y Vecindad) tomamos los mejores parámetros encontrados en los capítulos anteriores, los cuales son:

Migración		Vecindad	
Porcentaje	30*	Porcentaje	5*
Periodo o Epoca	1	Prob.de elección local	95
Topología	anillo+1*	Topología	anillo*
Parámetros con iguales valores en ambos modelos			
Número de máquinas		4	
Grafos (vértices, aristas, terminales, restric.)		cinco grafo(100,500,20,0..4)	
Condición de parada		300 generaciones	
Tamaño de la población (***SIZEPOP***)		30	
Probabilidad de mutación (***PMUT***)		0.00333	
Probabilidad de cruzamiento (***PCROSS***)		0.95	
Factor de escalado del fitness (***C***)		2	

* Los resultados no son afectados por este parámetro; se tomo este valor para fijar el modelo

Los resultados fueron los siguientes:



Observaciones:

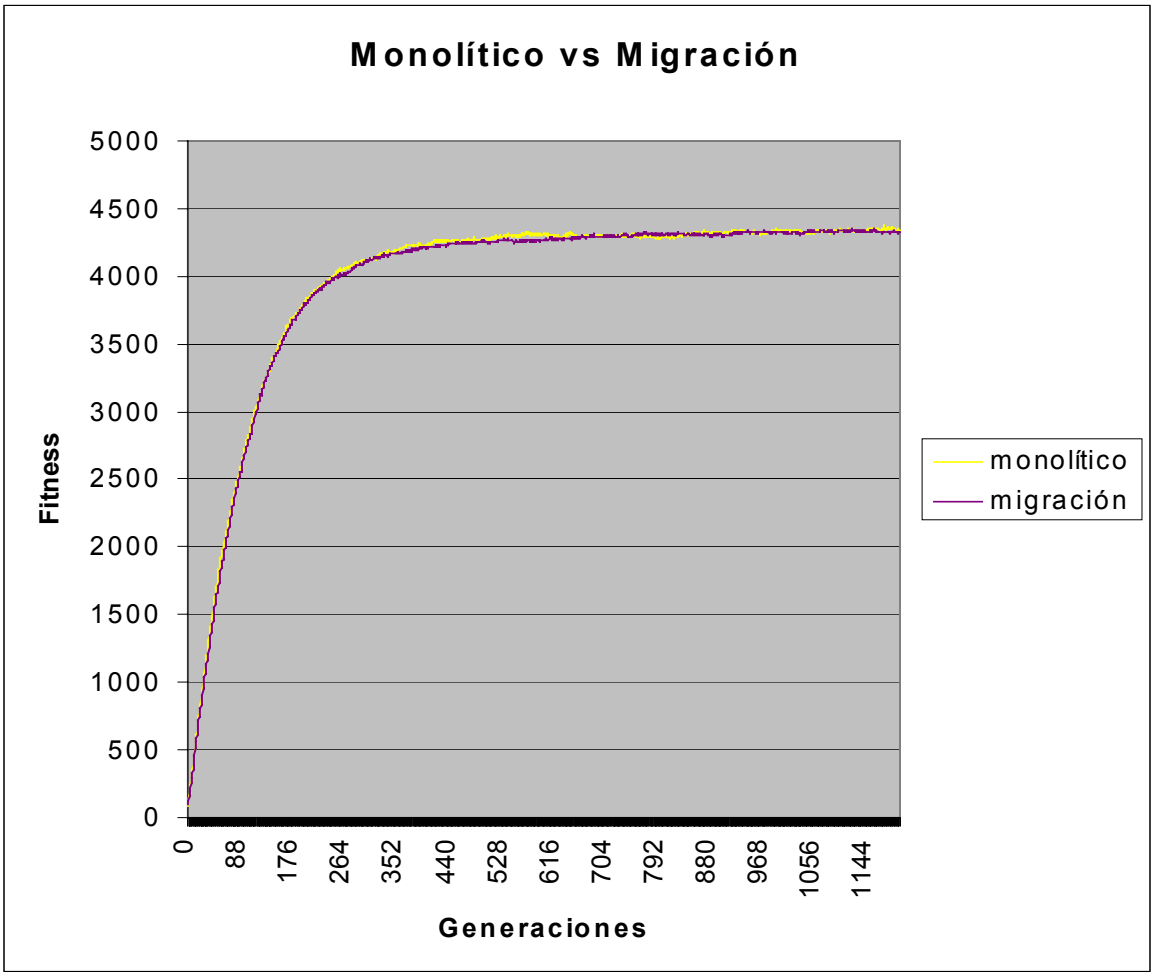
- No se aprecia que, habiendo elegido los mejores parámetros para cada modelo, uno sea superior al otro. Analizando la evolución de los resultados, el modelo de vecindad es superior al de migración en 165 de 299 generaciones, mientras que en las restantes 134 generaciones el modelo de migración es superior al de vecindad.
- No se aprecian diferencias significativas en cuanto al tiempo de procesamiento de ambos modelos.

10.3 Migración vs Monolítico

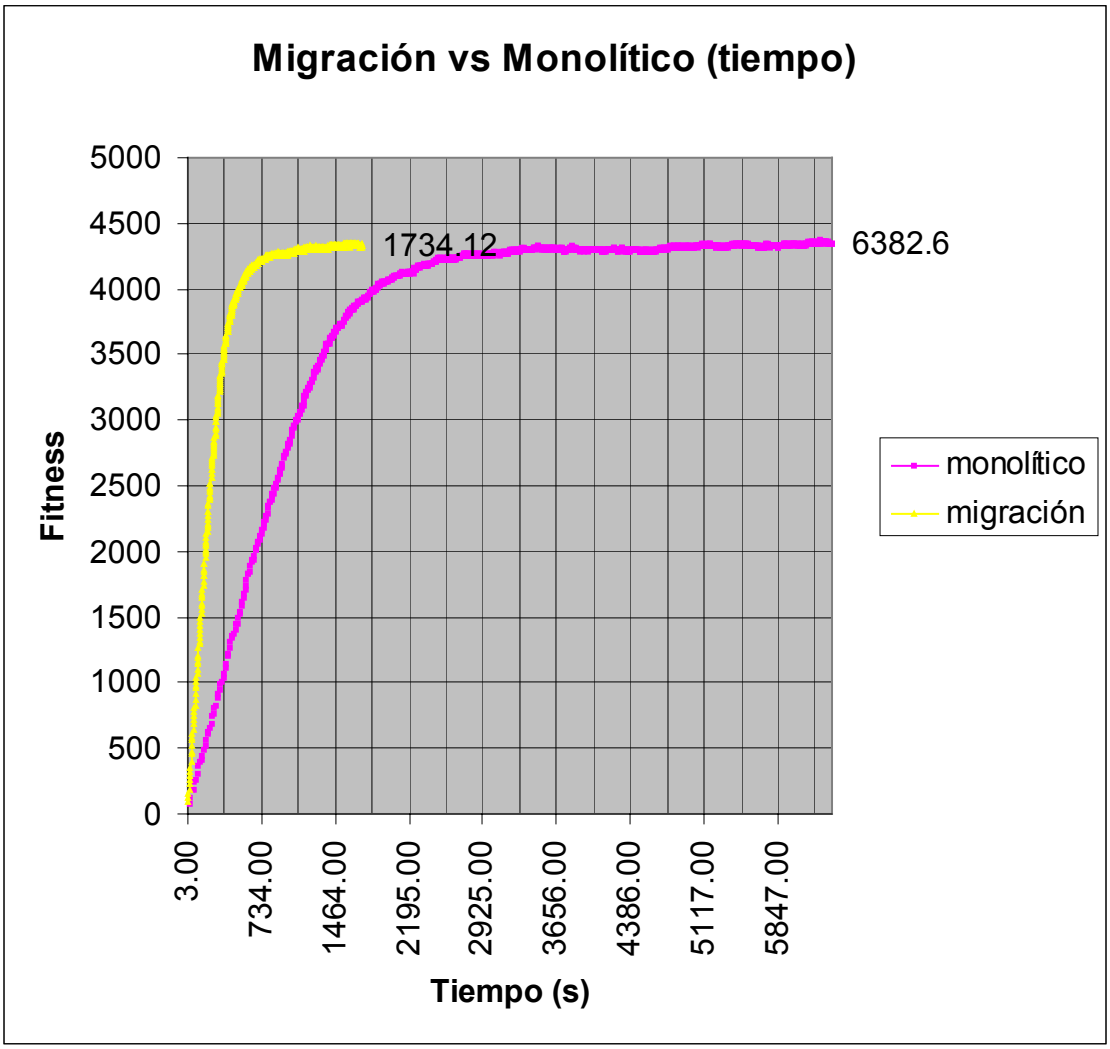
A continuación comparamos el modelo de migración con el modelo monolítico. Los parámetros utilizados en esta comparación son los siguientes:

Modelo	Migración	Monolítico
Número de máquinas	4	1
Tamaño de la población (***SIZEPOP***)	30	120
Porcentaje de migración	30	ignorado
Periodo o época	1	ignorado
Topología	anillo+1	ignorado
Parámetros con iguales valores en ambos modelos		
Grafos (vértices, aristas, terminales, restric)	cinco grafo(100,500,20,0..4)	
Condición de parada	300 generaciones	
Probabilidad de mutación (***PMUT***)	0.00333	
Probabilidad de cruzamiento (***PCROSS***)	0.95	
Factor de escalado del fitness (***C***)	2	

Los resultados fueron los siguientes:



La siguiente gráfica se muestra el tiempos consumido por cada modelo.



Observaciones:

- No se aprecia una superioridad significativa de un modelo sobre otro.
- En cuanto al tiempo, el modelo monolítico es 3.68 veces menos performante que el modelo de migración al dividir el problema en cuatro procesadores, a pesar de correr en un hardware 4.33 veces más rápido*.

.

* Como el algoritmo paralelo es sincrónico respecto a la evolución de las generaciones, este corre a la velocidad de la máquina mas lenta del cluster utilizado. La máquina lenta (Isidorito) no se encontraba operativa en el momento que se hicieron las pruebas del modelo monolítico por lo que éstas se hicieron en otra máquina del cluster (loki), la cual es 4.33 veces más rápida que la anterior.

Se presentan los resultados obtenidos con loki; sin embargo, para comparar adecuadamente se debería tener en cuenta la diferencia de velocidades entre las máquinas del cluster y la que corrió el modelo monolítico

11 Conclusiones y direcciones de investigación futura

11.1 Conclusiones

Se pueden obtener soluciones similares en calidad final al algoritmo original eligiendo adecuadamente el modelo de paralelización.

Se obtienen ganancias significativas en la velocidad de obtención de buenas soluciones al paralelizar. Las mejores instancias de los modelos paralelos implementados arrojan resultados similares tanto en calidad de soluciones como en la velocidad de obtención de las mismas.

El modelo de vecindad presenta una mayor sensibilidad frente a la variación de parámetros que el modelo de migración.

11.2 Investigación futura

A lo largo del proyecto fueron surgiendo diversas interrogantes que no fueron contestadas por falta de tiempo, recursos de hardware o su tangencialidad a los objetivos del mismo.

1. Estudio de otras heurísticas evolutivas. La utilizada (algoritmos genéticos) es probablemente la más conocida. Sin embargo existen otras heurísticas cuya base subyacente sigue siendo en general la evolución darwiniana, pero que utilizan otros enfoques o niveles de abstracción. En particular, sería interesante probar con las siguientes:

   - Programación evolucionaria (Fogel[23]).
   - Programación genética. (Koza[19]).
   - Estrategias de evolución (Rechenberg[20], Schwefel[21]).
   - Enjambres (Kennedy[22]). En este caso, el modelo subyacente no es el darwiniano, sino el de un "enjambre" de agentes idénticos que tratan de converger a una solución.

2. Estudio de los tiempos frente a la inclusión de nuevos nodos. ¿Qué tan escalables son las arquitecturas de paralelismo probadas? ¿Llega un punto en el que se obtienen mejoras marginales negativas?

3. Probar con más topologías:
   La poca cantidad de máquinas disponibles (4) restringe no sólo la cantidad de topologías a probar sino que las hace similares entre ellas. Por ejemplo, la distancia máxima entre un par de nodos en la topología de anillo + 1 es de dos saltos, mientras que en la topología totalmente conectada es de uno. Con más computadoras disponibles, la diferencia entre estas distancias crece linealmente. Como vimos, la elección de las topologías no afectó la calidad de las soluciones, con esta cantidad de máquinas..

4. ¿Son válidos los parámetros encontrados en los modelos paralelos al aumentar el número de nodos? Qué tan sensibles son a la variación en el tamaño del cluster?

5. Investigación de un modelo híbrido obtenido de la combinación los modelos de migración y vecindad. Se realizó la implementación de este modelo pero debido a gran cantidad de variables y la explosión combinatoria de posibles valores se realizaron pruebas limitadas para el mismo.

6. Para lidiar con la heterogeneidad de hardware en el cluster, se implementaron soluciones correspondientes a dos enfoques distintos:

    - Ignorarla: Bajar la velocidad del cluster a su mínimo común denominador. Este fue el enfoque utilizado para los experimentos finales, dado que nuestro objetivo principal era estudiar los distintos modelos de paralelismo en un cluster homogéneo, y que de esta manera nos independizábamos del hardware particular utilizado, haciendo los resultados generales y replicables en cualquier otro harware.

    - Aprovecharla todo lo posible. Para evitar que las máquinas más lentas atrasen la evolución de las generaciones en las más rápidas, se generan poblaciones de tamaño *variable* en cada generación, siendo proporcional el tamaño de la población a la velocidad del procesador. Luego de cada generación, se envía el tiempo demorado en procesarla al proceso controlador, que lo reenvía al resto del cluster. Con cada procesador sabiendo los tiempo de generación del resto, se puede ajustar localmente el tamaño de la población hasta que el tiempo empleado en procesar cada generación sea igual en todos los procesadores.

      Se corrieron pruebas con esta solución implementada, pero debido a la falta de reproducibilidad de los resultados, se eligió la alternativa anterior. Sin embargo, creemos que es un enfoque que merece un estudio detallado.

12 Bibliografía

[1] J. Krarup, “The generalized Steiner problem”, Nota no publicada (1978), citada por F. Robledo en [6].
[2] R. T. Chien, “Synthesis of a communication net”, IBM *J. Res. Develop.*, Vol. 3, pp. 325-334 (1960).
[3] R. E. Gomory y T. C. Hu., “Multi-terminal network flows”, *J. Soc. Indust. Appl. Math.* Vol 9, No. 4, pp 551-570 (1961).
[4] A. A. P. Klein y R. Ravi, “When trees collide: An approximation algorithm for the generalized Steiner Problem on Networks”, Reporte técnico CS-90-32, Brown University (1990).
[5] K. Steiglitz, P. Weiner y D. J. Kleitman, “The design of minimun cost survivable networks”, *IEEE Trans. On Circuit Theory*, CT-16, 4, pp. 455-460 (1969).
[6] F. Robledo, “Diseño Topológico de Redes”, Tesis de Maestría en Informática, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay, (2000).
[7] M.Charikar, C. Chekuri, T. Cheung, Z. Dai, A. Goel, S. Guha, M.Li, “Approximation Algorithms for Directed Steiner Problems”, Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) (1998).
[8] P. Winter, “Steiner Problems in Networks: A Survey”, BIT 25, pp.485-496 (1985).
[9] David E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning", Addison – Wesley, ISBN 0-20115767-5 (1989).
[10] L. Mettler y T. Gregg, “Genética de las poblaciones y evolución” Unión tipográfica de México. (1972).
[11] En línea al 25/07/02 en http://cs.felk.cvut.cz/~xobitko/ga/
[12] J. R. Koza, “Genetic Programming: On the Programming of Computers by Means of Natural Selection”, The MIT Press, Cambridge, Massachusetts (1992).
[13] Markus Schwehm, Paper: "Parallel Population Models for Genetic Algorithms" ().
[14] J. J. Grefenstette, *“Parallel adaptive algorithms for function optimization”*, Technical Report No. CS-81-19. Nashville, Vanderbilt University, Computer Science Department (1981).
[15] Henrik Esbensen, "Computing Near-Optimal Solutions to the Steiner Problem in a Graph Using a Genetic Algorithm", Proc. of The European Design an Test Conference, Paris, France. (1994).
[16] Donald E. Knuth, "The Art Of Computer Programming", Vol. 2, Addison -Wesley, (1969).
[17] L.R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, (1962)
[18] A.L. Baratz. *The complexity of the network flow problem.* Massachussets Institute of Technology, Laboratory for computer science, Cambridge, (1980), en línea al 25/07/02 en http://citeseer.nj.nec.com/524085.html.
[19] J.R. Koza. *Genetic Programming: On the programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press.
[20] I. Rechenberg. *Evolution Strategy*. En *Computational Intelligence – Imitating Life*, 137-159, Piscataway, NJ: IEEE Press.
[21] H. P. Schwefel. *On the evolution of evolutionary Computation*. En *Computational Intelligence – Imitating Life*, 137-159, Piscataway, NJ: IEEE Press.
[22]J. Kennedy y R. Eberhardt. *Swarm Intelligence*. ISBN 1-55860-595-9, 2001, Morgan Kaufman Publishers.
[23] D.B. Fogel *What is evolutionary computation?* IEEE Spectrum, Febrero 2000, 37(2), 26-32

13 Apéndices

13.1 Parámetros de entrada

La forma de invocar el sistema es la siguiente:

mpirun -machinefile <máquinas> -np <num_procs> gen2 [<parámetros>]

'mpirun' es un programa incluido en MPICH que permite ejecutar varios procesos gen2 en diferentes máquinas.

<máquinas> es el nombre del archivo que indica en que máquina deben correr los procesos. En cada linea de este archivo se escribe el nombre o dirección IP de la máquina.

El número de procesos está controlado por <num_procs>. MPICH decide en que máquina se ejecuta cada proceso. Si el número de procesos es M, los procesos se identificarán de 0 a M-1. Estos identificadores en MPICH se llaman ranks.

Los <parámetros> que le siguen a gen2 son opcionales. En caso de omitirlos se tomarán del archivo de configuración 'config' (ver 9.2). A continuación se detallan los parámetros.

-ng <grafo_entrada>
<grafo_entrada> Nombre del archivo del grafo de entrada. El formato de este archivo esta detallado en el punto 9.3.

-tp <modelo_paralelismo>
<modelo_paralelismo> Indica el modelo de paralelismo a utilizar. Existen tres tipos:
- 0: aislado
- 1: vecindad
- 2: migración.

-sg <grafo_resultado>
<grafo_resultado> Nombre del archivo del grafo solución. El formato de este archivo esta detallado en el punto 9.3.

-sl <log>
<log> Nombre de archivo donde se registra la bitácora del sistema.

Parámetros para migración:
Estos parámetros son aplicables cuando <modelo_paralelismo> es 2.
-fm <frecuencia>
<frecuencia> Frecuencia de migración, entero positivo.

-pcm <porcentaje>
<porcentaje> Porcentaje de individuos de la población local que migran.

-tm <topologia_migracion>
<topologia_migracion> Nombre del archivo de topología para migración.
El archivo debe tener el siguiente formato:
mig <origen> <destino>
.....
mig <origen> <destino>
mig <origen> <destino>
donde <origen> y <destino> identifican los 'ranks' de los procesos que envían y reciben los cromosomas respectivamente.

Parámetros para vecindad:
Estos parámetros son aplicables cuando <modelo_paralelismo> es 1.
-pcv <porcentaje_envio_barrios>
<porcentaje_envio_barrios> Numero real entre 0 y 1. 100% se representa con 1.

-pel <prob_eleccion_local>
<prob_eleccion_local> Numero real entre 0 y 1.

-tv <topologia_vecindad>
<topologia_vecindad> Nombre del archivo de topología para vecindad.
barrio <rank> <rank> ... <rank>
.....
barrio <rank> <rank> ... ... <rank>
barrio <rank> ... ... <rank>
donde cada linea indica cual es la relación de vecindad entre procesos representados por sus 'ranks'.

13.2 Archivo de configuración

El archivo de configuración debe llamarse "config" y el formato del mismo debe ser el siguiente:

```
-------------------------------------------------------------
# condiciones de parada
tiempo <tiempo>
max_gen <tope_gen>
epsion_parada <delta>

# parámetros del algoritmo genético
size_pop <tamaño>
prob_mut <p_mut>
prob_cross <p_cross>
factor_escalado <factor>

# generadores aleatorios
initrand <rank> <X0> <Y0> <A1> <A2> <C1> <C2> <M1> <M2>
.....................
initrand <rank> <X0> <Y0> <A1> <A2> <C1> <C2> <M1> <M2>
initrand <rank> <X0> <Y0> <A1> <A2> <C1> <C2> <M1> <M2>

# grafo de entrada
archivo_grafo <archivo>

# parámetros para el modelo de vecindad
barrio <id_proc> ... <id_proc>
.....................
barrio <id_proc> ... <id_proc>
barrio <id_proc> ... <id_proc>
porcentaje_envio_barrios <pev>
prob_eleccion_local <pel>

# parámetros para el modelo de migración
mig <id_proc_origen> <id_proc_destino>
.....................
mig <id_proc_origen> <id_proc_destino>
mig <id_proc_origen> <id_proc_destino>
frecuencia <fm>
comienzo <com>
procentaje <pcm>
-------------------------------------------------------------
```

Descripción de los valores

Condiciones de parada:
<tiempo> Tiempo en segundos, entero positivo.
<tope_gen> Número máximo de generaciones del proceso controlador, entero positivo.
<delta> Condición de convergencia de un proceso, real entre 1 y 1.999.
Se ignora la condición de parada si su valor es 0.

Parámetros del algoritmo genético:
<tamaño> Tamaño de la población.
<p_mut> Probabilidad de mutación.
<p_cross> Probabilidad de cruzamiento.
<factor> Factor de escalado de fitness.

Generadores aleatorios:
Cada línea tiene los parámetros del generador aleatorio para el proceso <rank>.
La relación que se sugiere mantener entre los valores esta descripta en el punto 4.5.

Grafo de entrada:
En este archivo esta el grafo de entrada del sistema. El formato de este archivo se describe en 9.3.

Parámetros para el modelo de vecindad:
<id_proc> Identificador de barrio, coincide con el rank de MPI.
<pev> Porcentaje de la población local que se envía a los barrios vecinos. Numero real entre 0 y 1. 1 representa 100%.
<pel> Probabilidad de elección de cromosoma de la población local.

Parámetros para el modelo de migración:
<id_proc_origen> Identificador del proceso que envía cromosomas.
<id_proc_destino> Identificador del proceso que recibe cromosomas.
<fm> Frecuencia de migración.
<com> Numero de generación a partir de la cual se comienzan las migraciones.
<pcm> Porcentaje de individuos de la población local que migran.

13.3 Formato de grafos

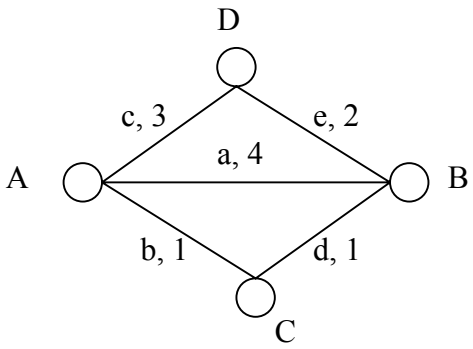
El sistema recibe y devuelve un grafo representado en el siguiente formato:

```
-------------------------------------------------------------------------------------------------------
vertice    <id_vertice>
vertice    <id_vertice>
……..
vertice    <id_vertice>
arista     <id_ari>   <id_vertice>    <id_vertice>    <costo>
……..
arista     <id_ari>   <id_vertice>    <id_vertice>    <costo>
arista     <id_ari>   <id_vertice>    <id_vertice>    <costo>
restriccion     <número>   <id_vertice>    <id_vertice>
restriccion     <número>   <id_vertice>    <id_vertice>
……..
restriccion     <número>   <id_vertice>    <id_vertice>
-------------------------------------------------------------------------------------------------------
```

<id_vertice>, <id_ari> son strings que identifican los vértices y aristas del grafo. <número> son enteros y representan las restricciones que se imponen entre el par de vértices terminales.

Ejemplo:

```
-----------------------------------------------------
vertice   A
vertice   B
vertice   C
vertice   D
arista    a    A    B    4
arista    b    A    C    1
arista    c    A    D    3
arista    d    B    C    1
arista    e    B    D    2
restriccion    1    A    D
restriccion    2    A    B
-----------------------------------------------------
```



El grafo de entrada puede ser leído desde un archivo especificado en la línea de comando con el parámetro ‘–ng’.

Como resultado, el sistema devuelve un grafo, correspondiente al cromosoma de mayor fitness obtenido en todas las generaciones. Este resultado se puede guardar en un archivo con el parámetro ‘–sg’.

13.4 Archivos de salida

13.4.1 Bitácora

El proceso controlador recibe reportes de las poblaciones (updates) respecto a los resultados que va obteniendo luego de cada generación. Estos reportes se pueden guardar en un archivo si se utiliza el parámetro '–sl' desde la línea de comando.

El archivo esta organizado con las siguientes columnas:
1. Tiempo transcurrido hasta el momento.
2. Identificador de proceso o población (rank).
3. Generación terminada
4. Fitness promedio
5. Fitness del mejor cromosoma de la población

De esta manera si en la bitácora tenemos la siguiente línea:

T id N prom máx

significa que el proceso controlador recibió en el tiempo ***T***, un update de la población ***id***, diciendo que culminó la generación ***N*** con un promedio de fitness ***prom*** y que el mejor cromosoma tiene fitness ***máx***.

Ejemplo:

```
3.54546   2    0    26.3333        73
3.54628   3    0    32.4           65
3.54691   3    1    61             117
7.0601    0    0    26.2333        68
7.06073   2    1    40.9667        116
7.06135   3    2    86.2667        127
7.06196   3    3    113.333        176
7.06259   3    4    144.333        221
10.5514   0    1    49.9           108
.............
```

13.4.2 Grafo resultado

El sistema devuelve el grafo correspondiente al cromosoma de mayor fitness obtenido en todas las generaciones. Este resultado se puede guardar en un archivo con el parámetro '–sg'. El formato del archivo es el mismo que el de grafo de entrada.

13.5 Generación de grafos aleatorios para el testeo de parámetros.

Para realizar las pruebas de los posibles parámetros, se debieron de generar grafos de Steiner de forma aleatoria. Estos grafos son más o menos complejos de acuerdo al número de aristas del mismo y de su cantidad de vértices terminales.

Se generaron grafos aleatorios de diferentes tamaños:

- pequeño: 50 vértices, 250 aristas, 10 vértices terminales, requerimientos de conexión aleatoriamente uniformes entre 0 y 3.
- mediano: 100 vértices, 500 aristas, 20 vértices terminales, requerimientos de conexión aleatoriamente uniformes entre 0 y 4.
- grande: 400 vértices, 2000 aristas, 80 vértices terminales, requerimientos de conexión aleatoriamente uniformes entre 0 y 5.

Debido a la gran cantidad de casos de prueba, se utilizan cinco grafos de tamaño grande.

Para generar los grafos aleatorios se utilizó una variación del algoritmo genético utilizado para resolverlos. Se comienza con cromosomas indicando el grafo vacío, con una o dos aristas seteadas al azar.

Se utiliza como medida de fitness la cantidad de requerimientos entre pares de vértices terminales satisfechos. Si el número de aristas en el grafo representado por el cromosoma es mayor que el pedido, la fitness desciende abruptamente a cero. Cuando se llega a un cromosoma cuyo grafo cumple todas las restricciones pedidas, se agregan aristas de forma aleatoria hasta cumplir el número de aristas pedido.

13.6 Documentación de prueba

13.6.1 Generación de casos de prueba

13.6.1.1 Casos de prueba para determinar parámetros

Los casos de prueba fueron generados el siguiente script perl, que en su salida estándar produce una línea por cada caso de prueba con el comando necesario para correrlo.

```
#!/usr/bin/perl


@grafos =("grafo_2_0",
          "grafo_2_1",
          "grafo_2_2",
          "grafo_2_3",
          "grafo_2_4");

@v_fm = (1,2,4,6,8,10);
@v_pcm = (0.01, 0.05, 0.10, 0.20, 0.30, 0.40, 0.50);
@v_topm = ("top_mig_1",
           "top_mig_3",
           "top_mig_4");

@v_pcv = (0.01, 0.05, 0.10, 0.20, 0.30, 0.40, 0.50);
@v_pel = (0.99, 0.95, 0.9, 0.80, 0.70, 0.60, 0.50);
@v_topv = ("top_vec_1",
```

```
                "top_vec_2");

#Primero, las corridas para migracion.

foreach $grafo (@grafos){
        foreach $topm (@v_topm){
                foreach $fm (@v_fm){
                        foreach $pcm (@v_pcm){
                                print "mpirun -machinefile machines -np 4 gen2 -ng problemas/$grafo -tp 1 -sg
resultados/$grafo -sl logs/mig-$grafo-$topm-$fm-$pcm -fm $fm -pcm $pcm -tm $topm\n";
                        }
                }
        }
}


# Ahora, las de vecindad

foreach $grafo (@grafos){
        foreach $topv (@v_topv){
                foreach $pcv (@v_pcv){
                        foreach $pel (@v_pel){
                                print "mpirun -machinefile machines -np 4 gen2 -ng problemas/$grafo -tp 2 -sg
resultados/$grafo -sl logs/vec-$grafo-$topv-$pcv-$pel -pcv $pcv -pel $pel -tv $topv\n";
                        }
                }
        }
}
```

13.6.1.2 Casos de prueba para comparación de modelos

13.6.2 Archivo de configuración utilizado en las pruebas

Se presenta el archivo de configuración de todas las pruebas. Si bien algunos parámetros (de migración, topologías, etc) son especificados en el archivo, estos son ignorados por el programa al estar dados en la línea de comandos.

```
// Cada linea son los parametros del generador aleatorio para cada poblacion.
// x0, y0 cualesquiera, ai-1 multiplos de 4, mi multiplos de 2, ci impares.

//                rank    x0      y0      a1      a2          c1          c2          m1          m2

initrand  0     77      91      65      17      23231231        87323   2147483648 2147483648
initrand  1     13      83      69      17      23231231        87323   2147483648 2147483648
initrand  2     19      29      13      17      23231231        87323   2147483648 2147483648
initrand  3     37      17      41      17      23231231        87323   2147483648 2147483648

tiempo          0
max_gen         1200
size_pop        30
prob_mut        0.00333
prob_cross      0.95
factor_escalado 2.0
epsilon_parada  0.0

/*****************************************************************************************/
/***********************          BARRIOS          *********************************/
/*****************************************************************************************/

// Informacion de la topologia de barrios.
// barrio  0 1 2 3
porcentaje_envio_barrios 0.05
```

```
prob_eleccion_local 0.95

/*****************************************************************************************/
/************************          MIGRACION          ***********************************/
/*****************************************************************************************/
// Información de migracion. Poblacion: 0..(NumProcesos - 1)
// mig <poblacion origen> <poblacion destino>
mig 0 1
mig 1 2
mig 2 3
mig 3 0

// Frecuencia de migracion
frecuencia 1

// Numero de generaciones a dejar pasar antes de comenzar las migraciones
comienzo 1

// Porcentaje de cada poblacion que participa en una migracion
porcentaje 0.03
```

13.6.3 Archivos de grafos utilizados en las pruebas.

Debido a su extensión, los mismos se entregan en formato electrónico.

13.6.4 Archivos de resultados de cada caso de prueba

Debido a su extensión, los mismos se entregan en formato electrónico.