UNIVERSIDAD
REY JUAN CARLOS

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Curso Académico 2009/2010

Proyecto de Fin de Carrera

ESTUDIO EXPERIMENTAL CON DIFERENTES OPERADORES DE CRUCE PARA LA RESOLUCIÓN DEL PROBLEMA DEL VIAJANTE

Autor: Jorge Acuña Acuña

Tutores: Abraham Duarte Muñoz
Juan José Pantrigo Fernández

Índice general

Resumen 3

1. Introducción 5

1.1. El Problema del Viajante 5
1.2. Los problemas NP 7
1.3. Algoritmos Genéticos 8

1.3.1. Codificación 10
1.3.2. Función de Evaluación 11
1.3.3. Criterios de Selección. 11
1.3.4. Cruce 12

1.3.4.1. *Partially-Mapped Crossover* (PMX) 13
1.3.4.2. *Order Crossover* (OX1) 15
1.3.4.3. *Order Based Crossover* (OX2) 17
1.3.4.4. *Alternating-Position Crossover* (AP) 18
1.3.4.5. *Position Based Crossover* (POS) 18

1.3.5. Mutación 19

2. Objetivos 21

2.1. Descripción del problema 21
2.2. Trabajos previos 21
2.3. Modelo de desarrollo 22

3. Diseño e implementación 23

3.1. Descripción del Algoritmo 23

3.1.1. Representación Cromosómica 26
3.1.2. Función de evaluación para el algoritmo 27
3.1.3. Función de selección 28
3.1.4. Descripción de los operadores genéticos 29

3.2. Descripción informática 36

3.2.1. Herramientas empleadas 37
3.2.2. Especificación de Requisitos 37
3.2.3. Análisis: Tareas 38
3.2.4. Estructura de datos 39
3.2.5. Descripción de funciones 39

4. Resultados 41

4.1. Análisis de resultados 48

5. Conclusiones 57

5.1. Lecciones aprendidas 58
5.2. Trabajos futuros 58

Bibliografía 58

Resumen

El Problema del Viajante (*TSP*) es uno de los problemas NP más conocidos, fácil de comprender pero de difícil solución, y los algoritmos genéticos son utilizados en la resolución de los mismos. Este tipo de algoritmos simulan el proceso evolutivo de la naturaleza y nos permiten obtener resultados óptimos en tiempos razonables.

Ante tales afirmaciones, el presente proyecto intentará realizar un análisis del comportamiento de este tipo de algoritmos ante diferentes situaciones, así como una comparativa con diferentes operadores de cruce, todo ello aplicado en la resolución del *TSP*.

Para ello, primeramente se presentarán los conceptos fundamentales que nos permitirán desarrollar nuestra aplicación, para posteriormente pasar a detallar la implementación de nuestra herramienta con el lenguaje de programación MATLAB.

Por último se presentarán los resultados y se analizarán, teniendo como criterios de comparación: el aumento del número de generaciones, el aumento del número de individuos de la población, los diferentes operadores de cruce implementados, los tiempos obtenidos etc.

Capítulo 1

Introducción

En este primer capítulo se presenta una breve introducción acerca del Problema del Viajante (*Traveling Selesman Problem*,TSP). Además se introducen los Algoritmos Genéticos, detallando sus partes fundamentales, su funcionamiento y utilidad. La intención de este capítulo es, en suma, presentar los elementos básicos de nuestro proyecto.

1.1. El problema del viajante

El Problema del viajante, también conocido como *El Problema del Viajante del Comercio* y como *TSP* (*Traveling Selesman Problem*), es uno de los problemas de Optimización Combinatoria más conocido. Consiste en:

"Encontrar entre un conjunto de ciudades, una ruta que partiendo de una ciudad y regresando a ella, recorra todas las ciudades por el camino más corto."

Es un problema típico de cálculo y optimización de rutas y, a pesar de ser relativamente fácil de entender, es muy complejo de resolver.

Una definición más formal del problema es la siguiente:

Sean N ciudades y una matriz de costos donde C_{ij} (con i,j=1,...,n), se trata de encontrar la permutación $\pi=(\pi(1),...,\pi(n))$ con menor coste asociado. Es decir:

$$\pi = \min \quad (c_{\pi}(1)\pi(n) + \sum_{i=1}^{n-1} c_{\pi}(i)\pi(i+1))$$

El problema debe de tener en cuenta las siguientes restricciones básicas:

- ✓ Tiene que pasar por todas las ciudades.
- ✓ Cada ciudad es visitada sólo una vez, excepto la ciudad origen.

Según la teoría de grafos, el problema consiste en encontrar un ciclo de longitud mínima en un grafo no dirigido que comience y termine en el mismo vértice y pase exactamente una vez por cada uno de los vértices restantes. Es decir, se trata de encontrar un ciclo hamiltoniano de coste mínimo. Por ello, el problema se modela mediante un grafo, donde las ciudades son los vértices y la distancia entre ciudades se corresponde con el valor o peso asociado las aristas del grafo. Como todas las ciudades están unidas entre sí, en nuestra representación tendremos un grafo completo. En la Figura 1.1 se muestra un ejemplo con 7 ciudades.

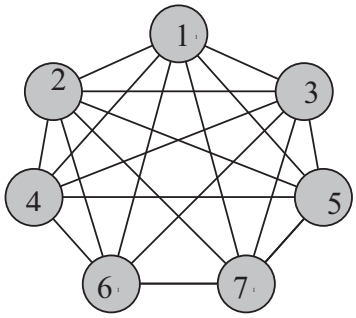


Figura 1.1: Grafo completo no dirigido con 7 vértices y 21 aristas

Sea un grafo completo $G = \{V, E\}$ con un conjunto de vértices $V = \{i, \ldots, j\}$ y un conjunto de aristas $E = \{[i, j], \text{ donde } i, j \in V, i \neq j\}$. Se definen C_{ij} como el coste de ir de i a j y la variable X_{ij} una variable entera tal que:

$X_{ij} = 1$ si se pasa por la arista $[i,j]$

$X_{ij} = 0$ si no se pasa por la arista $[i,j]$

El modelo matemático basado en la teoría de grafos para el TSP sería el siguiente:

$$\varpi^{TSP} = mín \sum_{e \in E} C_e \cdot X_e$$

Sujeto a:

$$(1) \sum_{e \in \delta(v_{ij})} x_e = 2 \quad \forall \; v_i \in V$$

$$(2) \sum_{e \in \delta(S)} x_e \geq 2 \quad \forall \; S \subset V$$

$$(3) \; x_e \in \{0,1\} \quad \forall \; e \in E$$

La restricción (1) obliga a que el número de aristas incidentes en cada vértice sea igual a 2. Con el fin de evitar subciclos, las restricciones de eliminación de subciclos (2) establecen que un subconjunto $S \subset V$ debe estar conectado con su complementario por al menos dos aristas del ciclo.

El número de posibles soluciones vendría determinado por el factorial de las n ciudades ($n!$), lo cual implica que, para un número de ciudades elevado, necesitaría de un tiempo de cómputo exagerado.

Hasta la fecha no se ha encontrado un algoritmo que resuelva este problema de manera polinomial, ya que con los algoritmos conocidos el número de soluciones crece de manera exponencial a medida que crece el número de ciudades. Por ello pertenece a un conjunto de problemas conocido como NP. Como los algoritmos clásicos no han logrado solucionarlo, tenemos que utilizar otras técnicas para intentar resolverlo o acercarnos a una solución de la mayor calidad posible, en un tiempo de ejecución razonable.

1.2. Problemas NP

Los problemas se clasifican según la complejidad computacional de los algoritmos que los resuelven. Según la teoría de la complejidad computacional se dividen en dos clases *P* y *NP*.

Los problemas de la clase *P* son aquéllos que se resuelven en un tiempo polinomial, es decir, se puede determinar una fórmula polinómica que relacione el tiempo de ejecución y el tamaño de la entrada del algoritmo. En cambio, existe un conjunto de problemas difíciles de resolver, para los que no se ha encontrado una solución en un tiempo polinomial. Éstos son los conocidos como *NP*. Los tiempos de ejecución también son una de las diferencias de estos tipos de problemas, por un lado los *P* tienen tiempos de ejecución razonables, mientras que los *NP* no.

Para muchos problemas *NP* se ha podido averiguar de forma polinomial si un valor se encuentra entre las posibles soluciones del problema. Esto nos lleva a la conclusión de que los problemas *P* también son problemas *NP*, ya que siempre podremos comprobar de forma polinomial si un valor es solución del problema.

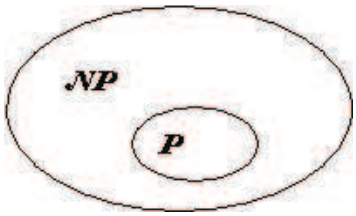


Figura 1.2: Relación entre los problemas *P* y *NP*

1.3. Algoritmos Genéticos

Los Algoritmos Genéticos (GA) son métodos adaptativos que se utilizan en la resolución de problemas de búsqueda y optimización complejos, aplicando técnicas que simulan la evolución natural de los seres vivos.

Los Algoritmos Genéticos fueron introducidos por John Holland en torno al año 1970, el cual se inspiró en el proceso de la evolución de las especies observado en la naturaleza para intentar resolver ciertos problemas. Al principio, sus ideas no tuvieron gran acogida en la comunidad científica, pero fue en la Universidad de Michigan donde comenzó a desarrollar sus ideas y donde sus estudios dieron los primeros frutos. Los puntos fundamentales de su investigación fueron imitar los procesos de evolución y adaptación en la naturaleza, y diseñar programas informáticos que recogieran esos conceptos.

En estos últimos años, los Algoritmos Genéticos han experimentado un gran auge. El mayor crecimiento se ha producido en áreas donde no existen técnicas rápidas y precisas para resolver sus problemas más complejos. Actualmente encontramos estos algoritmos aplicados en áreas como la Medicina, las Telecomunicaciones, la Química, la Informática, etc. También se utilizan para encontrar soluciones aproximadas a los problemas difíciles de resolver, típicamente *NP*, como puede ser el problema del viajante (TSP), el problema de la mochila (*Knapsack Problem*), el problema de asignación de recursos (*Asignament Problem*), etc.

El gran éxito de estos algoritmos radica en que, aunque no garantiza la solución optima, proporciona una solución que es aceptablemente buena en un tiempo también aceptable.

En los Algoritmos Genéticos, el espacio de búsqueda del problema que se quiere resolver es representado con una colección de individuos llamados cromosomas. Cada cromosoma es una de las posibles soluciones del problema y donde se encuentra codificada la información. Esta información se representa como un conjunto de parámetros conocidos como genes. El objetivo del algoritmo es encontrar el individuo con mejor adaptación al medio. Para ello se utiliza una función que evalúa la calidad ("*fitness*") del individuo. Esta función nos va a determinar cómo de bueno es cada cromosoma y, por lo tanto, debe estar relacionada con la función objetivo del problema que queremos resolver.

La información de los cromosomas irá cambiando de unas generaciones a otras. Con estas generaciones lo que se pretende es crear poblaciones nuevas que contengan individuos con un material genético nuevo, por lo que necesitaremos ciertas operaciones para que el cambio de la información en los cromosomas se produzca, ya que en ellos radica la evolución. Los dos métodos utilizados son conocidos como **cruce** y **mutación.**

El cruce se utiliza para la recombinación de información entre dos individuos y consiste en el intercambio de ciertos genes entre dos cromosomas, llamados padres, dando como resultado dos cromosomas con información nueva llamados hijos. En general, los individuos que se cruzan son los más adaptados al medio, por lo tanto, necesitamos un criterio de selección. La mutación en cambio consiste en la generación de un hijo a partir de un solo padre cambiando uno o más genes al azar.

Resumiendo, podemos decir que los componentes básicos de un Algoritmo Genético son los siguientes:

- Una representación cromosómica.
- Una población inicial de individuos.
- Una función de evaluación de los individuos.
- Un criterio de selección.
- Un criterio de recombinación.
- Un criterio de mutación.

A continuación, veremos en formato de pseudocódigo la estructura básica de un Algoritmo Genético.

```
BEGIN /*Comienzo del algoritmo
        Generar una población inicial;
        While (número de generación<=número
        generaciones máximo)
          {
                Evaluar Población;
                Selección de individuos;
                Cruzar;
                Mutar;
                Insertar hijos en la nueva población;
                Numero de generaciones++;
          }
```

Figura 1.3: Pseudocódigo de Algoritmo Genético

El uso de estos algoritmos requiere la definición de todos los componentes expuestos anteriormente, por lo que en el resto de la sección describiremos más detalladamente cada uno de ellos.

1.3.1. Codificación

La codificación es el primer paso en el diseño de un Algoritmo Genético. La elección de la codificación es un paso muy importante porque permite definir cada individuo de la población y además determina los operadores genéticos del algoritmo. La mayoría de las veces, una codificación correcta es la clave para obtener un buen resultado.

Al principio, para codificar los genes que forman un cromosoma se utilizaban **Strings Binarios**, es decir ristras de 0's y 1's.

$$X=(x_1, x_2, x_3, \ldots, x_n) \qquad xi \in (0, 1)$$

Ej: x= 1001011101010001111000101 01

Esta codificación permite tener operadores de recombinación muy sencillos y además la convergencia del algoritmo está prácticamente probada, pero a veces nos limita la representación de la semántica de ciertos problemas. Un ejemplo de esta limitación es el

problema que se trata en este documento, donde la representación de las rutas resulta bastante complicada.

Para solucionar este problema se recurren a otros tipos de codificación, por ejemplo, una basada en números naturales. Véase el siguiente ejemplo.

$$X=(x_1, x_2, x_3, \ldots, x_n) \qquad xi \in [1, 9]$$

$$X= 1\ 3\ 2\ 4\ 9\ 7\ 8\ 5\ 6$$

1.3.2. Función de Evaluación

Esta función determina la calidad de cada cromosoma de la población; es decir, la calidad de la solución. La función de evaluación es propia de cada problema.

Su objetivo es asignar un valor a cada individuo proporcional a su calidad. Esta calidad será utilizada para seleccionar los individuos que se cruzan, y sirve de medida penalizadora de aquellos individuos peores, ya que éstos no sobreviven.

La función de evaluación también influye en la convergencia del problema. Si las diferencias de calidad entre los individuos mejores y los peores son muy grandes el algoritmo va a converger más rápidamente y, en cambio, si esas diferencias son pequeñas se produce un estancamiento del algoritmo, por lo que se tiene que buscar una función de evaluación que haga que el problema converja razonablemente y evite el estancamiento del mismo.

1.3.3. Criterios de Selección

Una vez que los individuos son evaluados, se necesita un criterio de selección para escoger aquellos que van a producir descendientes. El criterio de selección debe favorecer la selección de los cromosomas con mayor calidad para que los individuos de la siguiente población sean mejores, y así ir obteniendo una mejor solución del problema que se quiere resolver. Existen múltiples criterios de selección pero los más comunes son los siguientes:

- **Método de la ruleta**: Este método, que es el más habitual, selecciona a los padres de forma aleatoria ponderada, de modo que se favorezca la reproducción de los de mejor calidad. Para ello, se asigna a cada individuo una probabilidad que es proporcional a su calidad o "*fitness*", normalmente mediante la siguiente función:

$$\Pr(h_i) = \frac{fitness\,(h_i)}{\sum_{j=1}^{n} fitness\,(h_j)}$$

  Una vez que cada individuo tenga asignada su probabilidad, se genera un número aleatorio *r* entre 0 y 1. Por último, se toma el primer individuo cuya probabilidad sumada a la de los individuos que lo preceden sea mayor o igual a *r*.
- **Basado en Rango**: Básicamente este método consiste en ordenar los individuos según su "*fitness*", eliminar los *N* peores de la población y sustituir éstos con los hijos de los *N* mejores con otros miembros de la población.
- **Selección de torneo**: Este método consiste en elegir un número de individuos al azar, de los cuales el mejor se reproduce sustituyendo al peor.

1.3.4. Cruce

Consiste en el intercambio de material genético entre dos individuos de la población. Este intercambio de información genera dos hijos que contiene material genético de ambos padres. Los padres pueden descender de los mismos progenitores, pero se tiene que tener cierto cuidado porque si esto se produce muy a menudo pueden propagarse genes no deseados.

Existen muchos tipos de cruce pero los más habituales son los siguientes.

- **De un punto**: Donde se selecciona un punto del cromosoma aleatoriamente, y se intercambia entre los padres, la cadena de genes comprendida desde el punto seleccionado hasta el final de cada cromosoma (Ver figura 1.4).

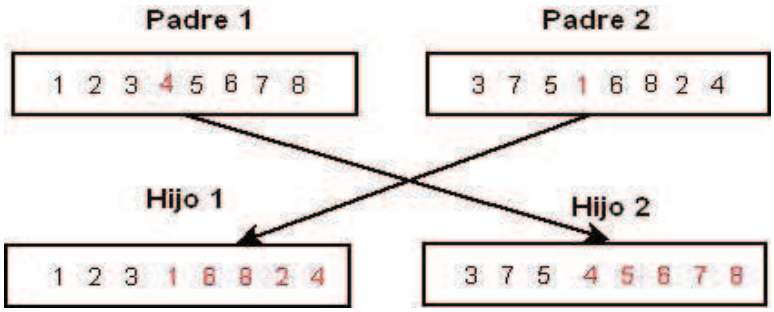


Figura 1.4: Representación de cruce de un punto

- **De dos puntos**: Donde se seleccionan dos puntos del cromosoma y se intercambian los genes de los padres comprendidos entre ambos puntos (Ver figura 1.5).

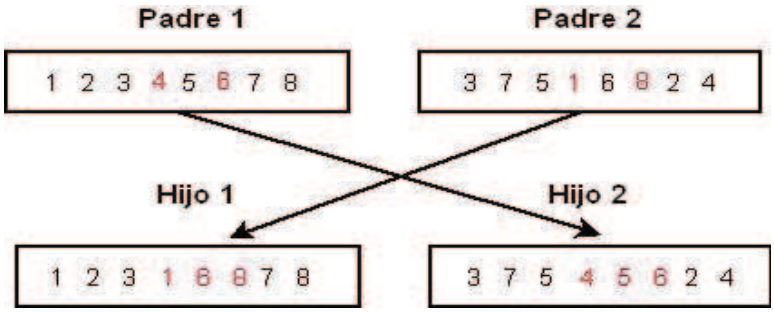


Figura 1.5: Representación de cruce de dos puntos

- **Uniforme**: Este cruce forma los hijos con genes que se van escogiendo aleatoriamente de los padres.
- Otros tipos más sofisticados que, en general, son dependientes del problema, pueden ser **PMX**, **OX1**, **VR**, etc.

A continuación se exponen los cruces más utilizados en el Problema del Viajante.

1.3.4.1. *Partially-Mapped Crossover* (PMX)

Fue sugerido por Golberg y Lingle (1985). Se basa en las posiciones de los genes y en el valor de los mismos. Este tipo de cruce consiste en el intercambio de un cierto número de genes entre los padres, lo que provoca que el resto de la información también cambie a la hora de crear los descendientes.

Los padres intercambian el mismo número de genes y éstos se encuentran situados en las mismas posiciones dentro de sus respectivos progenitores. Este conjunto de genes se conoce como patrones. Aplicándolo a la resolución del TSP, supónganse las siguientes rutas para el problema de las 8 ciudades:

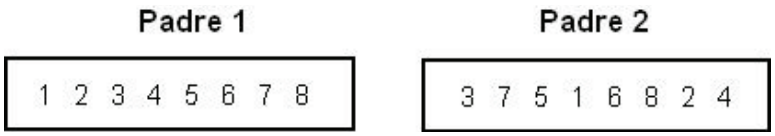


Figura 1.6: Representación de los padres

El cruce selecciona dos valores que indican el principio y el final de los patrones que vamos a intercambiar. En nuestro ejemplo los valores son la posición 4 y 6 (Véase la figura 1.7).

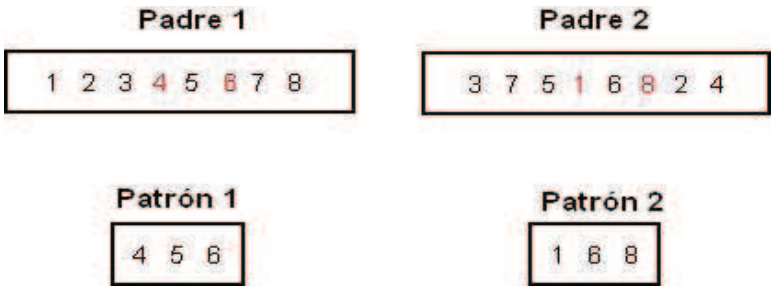


Figura 1.7: Patrones seleccionados

Una vez seleccionados los patrones se produce el intercambio entre los padres.

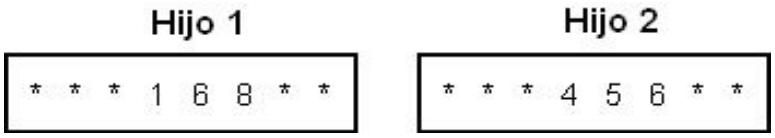


Figura 1.8: Representación de los hijos con los patrones intercambiados

Y por último se procede a introducir el resto de la información genética. Nos encontramos dos casos dependiendo de si el gen a introducir está o no en el patrón introducido. Si el gen no se encuentra en el hijo se introduce ese gen en la misma posición que tiene en el progenitor (Véase figura 1.9).

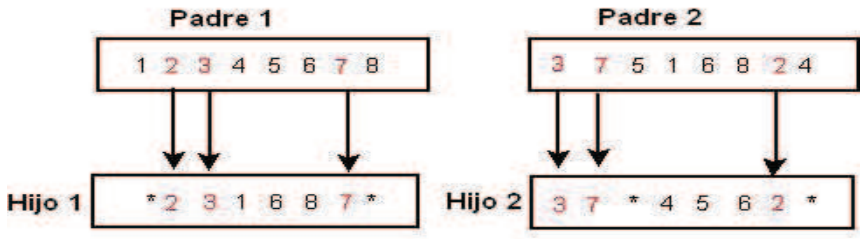


Figura 1.9: Intercambio de información entre los padres

Pero si el gen a introducir se encuentra en el hijo, se procede de la siguiente manera:

1. Ver en que posición se encuentra el gen en el hijo.
2. Tomar esa posición.
3. Ahora el gen a introducir es el que se encuentra en la posición tomada en el punto 2. Si está nuevamente en el patrón se vuelve a 1, si no, se coloca ese gen en el hijo en la posición del gen inicial (Véase figura 1.10)

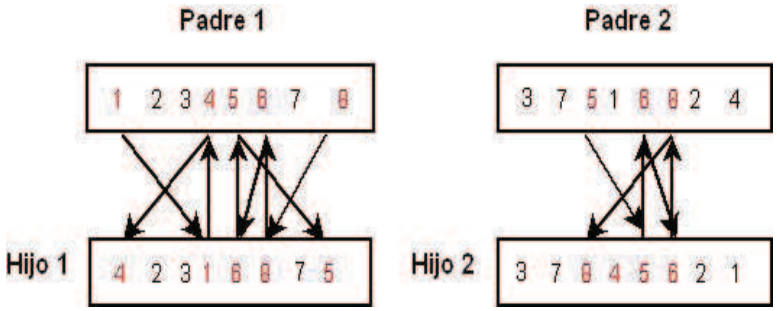


Figura 1.10: Resultado del cruce PMX

1.3.4.2. *Order Crossover* (OX1)

Fue sugerido por Davis (1985). Este tipo de cruce está basado en el orden en el que aparecen los genes y no en sus posiciones dentro de los individuos. Para crear un descendiente se copia en él un conjunto de genes de un padre (patrón) y el resto de las ciudades se copian en el mismo orden en el que aparecen en el otro padre. Al igual que con PMX los patrones contienen el mismo número de genes y éstos están en las mismas posiciones dentro de sus progenitores. Véase el siguiente ejemplo y supónganse las siguientes rutas

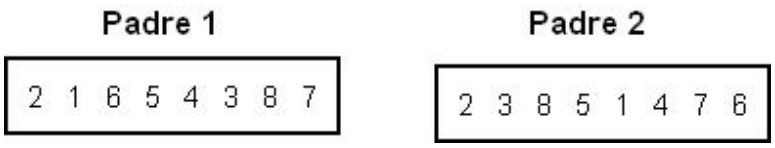


Figura 1.11: Representación de las rutas para el ejemplo

Primero se seleccionan aleatoriamente dos valores que indican el principio y el final de los patrones. Por ejemplo las posiciones 3 y 6.

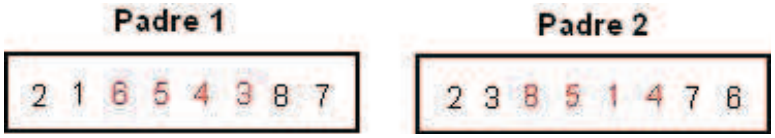


Figura 1.12: Representación de los cromosomas con los patrones seleccionados

A continuación, se copian estos genes dentro de los descendientes, intercambiándose entre los padres. Como lo estamos aplicando al problema del TSP, se respeta la primera posición del cromosoma ya que es nuestra posición origen (Véase figura 1.13).

Figura 1.13: Descendientes con los patrones intercambiados

Y por último se copia el resto de los genes en el orden en el que aparecen en el otro padre. Para ello se recorren los genes de los padres y si el gen no se encuentra en el hijo lo introducimos en él. Se introducen a partir del patrón del hijo y, cuando se llegue al final, se continúa en la segunda posición (Véase figura 1.14).

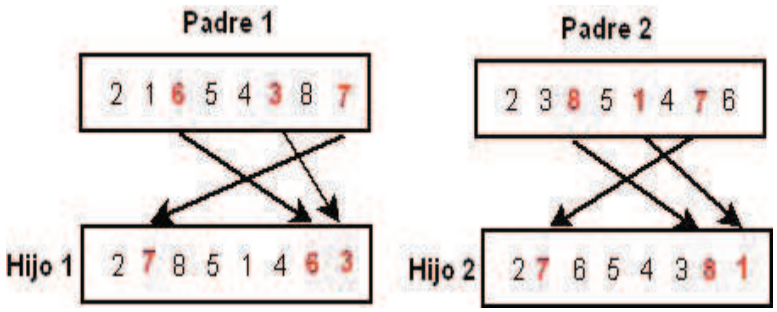


Figura 1.14: Intercambio del resto de la información de los progenitores

1.3.4.3. *Order Based Crossover* (OX2)

Fue propuesto por Syswerda en 1991. Para crear los descendientes este tipo de cruce elige aleatoriamente unas posiciones y, el orden en el que aparecen las ciudades en esas posiciones en un padre, es impuesto en el otro padre. Supóngase que tenemos los siguientes progenitores.

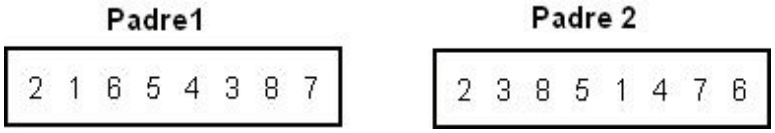


Figura 1.15: Representación de las rutas para el ejemplo

Primeramente se eligen de forma aleatoria unas posiciones, en este caso elegiremos 3.

Figura 1.16: Elección de las posiciones aleatoriamente

Como vemos en el Padre1 los genes que se encuentran en esas posiciones son el 1, el 4 y el 3. Lo siguiente será buscar en el otro padre esos genes e imponerle el orden en el que aparecen estos en el Padre1. Como resultado en el Padre2 después de realizar el cruce las ciudades 3, 1, y 4 aparecerán con el orden 1, 4 y 3.

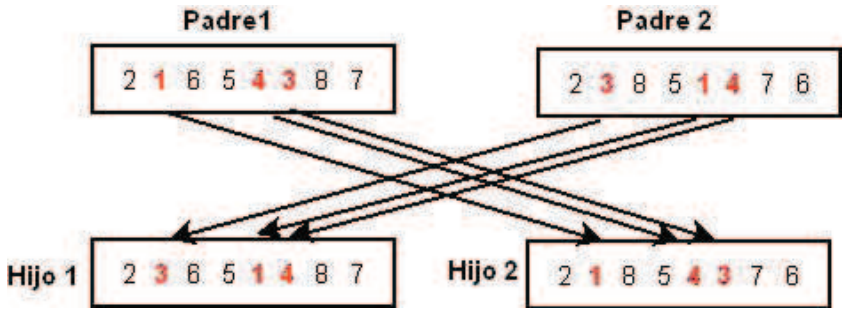


Figura 1.17: Intercambio de la información entre los padres

1.3.4.4. *Alternating-Position Crossover* (AP)

Fue propuesto por Larrañaga en 1996 y simplemente crea un descendiente alternado las ciudades de los padres. Para ello se introduce alternativamente un gen de cada progenitor hasta haber completado el descendiente. Cuando el gen a introducir ya se encuentra en el descendiente se pasa al siguiente del mismo padre y así se evita tener genes repetidos.

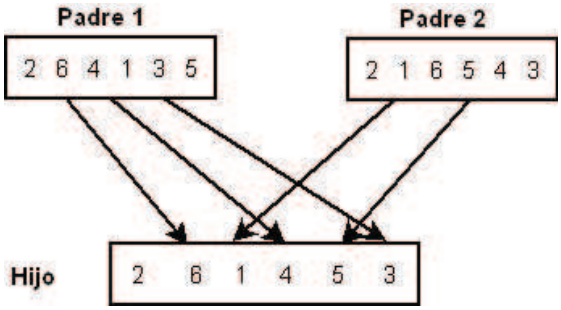


Figura 1.18: Representación cruce AP

1.3.4.5. *Position Based Crossover* (POS)

Fue propuesto por Syswerda en 1991 y se basa en las posiciones de los genes. Para crear los descendientes se elige un número de posiciones aleatorias y se introducen los genes que se encuentran en esas posiciones. Después de tener introducidos esos genes se introduce el resto de los genes del otro progenitor, empezado desde el principio e introduciendo sólo los genes que no se encuentran en el descendiente.

En el siguiente ejemplo se introducen los genes que se encuentran en las siguientes posiciones: 2 5 y 7.

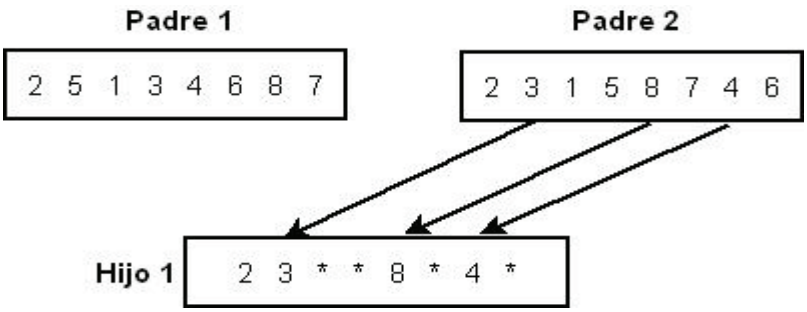


Figura 1.19: Primera fase del cruce Pos

Y para terminar se introduce el resto de genes del otro progenitor de la siguiente manera:

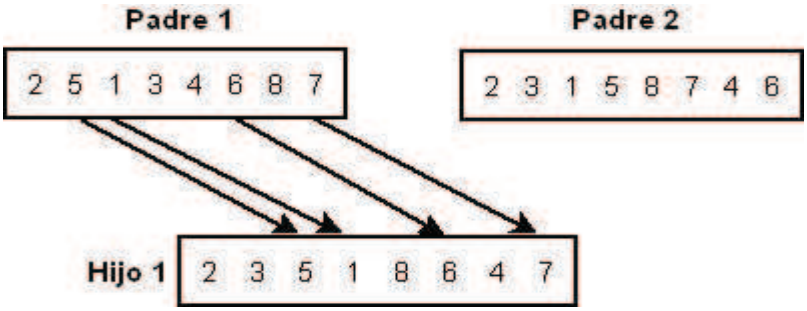


Figura 1.20: Última fase del cruce Pos

1.3.3. Mutación

Es el otro operador genético por excelencia. Produce un nuevo hijo a partir de un solo padre, modificando alguno de sus genes al azar. Normalmente se utiliza después del cruce y su objetivo es introducir diversidad en la población. Al igual que ocurre en la naturaleza, las mutaciones no deben ser muy habituales.

Existen múltiples operadores de mutación. Los más utilizados suelen ser los siguientes:

- ***Order-Based:*** Esta mutación consiste en elegir dos ciudades al azar e intercambiar sus posiciones.
- ***Sublist-Based*:** Consiste en seleccionar dos posiciones en un cromosoma e invertir las ciudades que se encuentran entre esas dos posiciones
- ***Partial-Gene:*** Selecciona aleatoriamente un gen del cromosoma y lo modifica .

Para realizar las mutaciones se define una probabilidad que llamamos probabilidad de mutación. Una vez se tenga esa probabilidad definida se recorren uno a uno los genes de los

padres. Para que un gen mute se genera un número al azar y, si ese número es menor que la probabilidad de mutación, entonces el gen cambia.

Capítulo 2

Objetivos

2.1. Descripción del problema

Este trabajo se centra en hacer una evaluación de la metodología de los Algoritmos Genéticos (GA) y en la resolución del *Problema del Viajante (TSP),* además de estudiar el comportamiento de los operadores de cruce *PMX, OX1, OX2, POS y AP*.

Con esto se persigue describir cuantitativamente el comportamiento del Algoritmo Genético (GA) ante diferentes situaciones, como: número de individuos *(cromosomas),* número de generaciones y operadores de cruce.

Los objetivos fundamentales son dos:

- La creación e implementación de una herramienta que permita parametrizar y lanzar nuestro Algoritmo Genético aplicado al TSP.
- El estudio de los datos obtenidos con nuestra herramienta para los diferentes operadores de cruce.

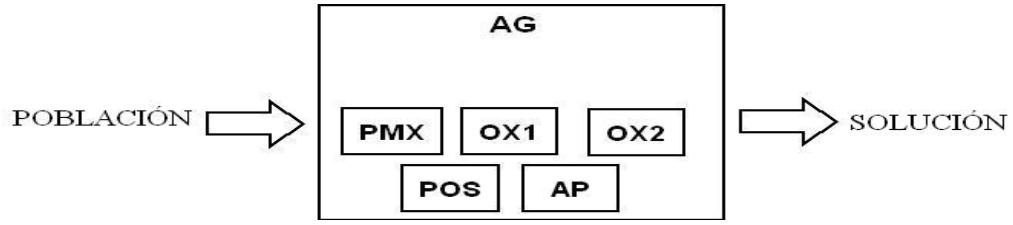


Figura 2.1: Descripción del Problema

2.2. Trabajos previos

Para realizar este proyecto ha sido necesaria la realización de trabajos previos. Se comenzó por una fase de documentación, en la que se recopiló toda la información necesaria acerca de los Algoritmos Genéticos, de su ámbito de aplicación, de su estructura básica, etc. Primeramente, se estudió un documento titulado "Genetic Algorithm Approaches to Job Shop

Scheduling Problems (Dept. of Computer Science Ansystem Engineering, Kyushu Institute of Technology, Iizuka Japan and Corea)", donde se tuvo un primer acercamiento a los algoritmos genéticos, su estructura, funcionamiento y aplicaciones al problema del Scheduling.

Otros de los documentos estudiados fue una introducción a los algoritmos genéticos hecha por el profesor Jesús M. De la Cruz y José A. López Orozco, haciendo algunas pruebas con un algoritmo genético implementado por ellos. De esta forma, nos familiarizamos con los genéticos y el lenguaje MATLAB.

Una vez que se adquirieron unas nociones básicas sobre estos algoritmos y sobre MATLAB, se abordó la fase de documentación acerca del problema del viajante.

2.3. Modelo de desarrollo

En el desarrollo un proyecto software se siguen una serie de etapas, conocidas como modelo de desarrollo. Un modelo de desarrollo establece un orden en la creación de un proyecto y una serie de requisitos a cumplir en cada una de las etapas de las que se compone. El modelo de desarrollo que se sigue en la realización de este proyecto es un desarrollo en espiral. Es decir, el proyecto ha sido creado a través de múltiple repeticiones del ciclo de vida.

Con este modelo de desarrollo, al finalizar cada ciclo se obtiene un prototipo que nos sirve para evaluar si los requisitos especificados se van cumpliendo. Cada una de las iteraciones de un desarrollo en espiral se puede dividir en las siguientes fases:

- **Análisis de requisitos**. Fase en la que se determinan los objetivos del proyecto.
- **Diseño y planificación**. Donde se describe como el sistema va a satisfacer los requerimientos y donde se codifica.
- **Pruebas**. El objetivo es comprobar que efectivamente se ha conseguido cumplir con los objetivos que se plantearon en el análisis de requisitos.
- **Planificación del siguiente ciclo,** que se elabora en base a los resultados obtenidos en las pruebas anteriores. Si los resultados han sido satisfactorios, la siguiente fase se planteará como un incremento de la funcionalidad del sistema, en caso contrario se planteará como un incremento de la robustez y consistencia del mismo.

Capítulo 3

Diseño e implementación

Este capítulo se dedica a describir el diseño de la herramienta creada para nuestro estudio, que abarca, el algoritmo genético utilizado, la función de evaluación, los operadores de cruce, etc. Además, se detalla la implementación informática de los mismos.

3.1. Descripción del Algoritmo

Una vez adquiridos los conocimientos básicos sobre el *Problema del Viajante (TSP)*, y conocidos en profundidad los Algoritmos Genéticos (*GA*), nos enfrentamos a la construcción de la herramienta que vamos a utilizar para nuestro estudio experimental.

A la hora de enfrentarse al diseño de nuestro algoritmo, se tuvieron que tener en cuenta muchos aspectos para llegar a una correcta solución. Para ello se estudiaron diferentes propuestas de diseño de Algoritmos Genéticos.

Una vez elegido el algoritmo genético 'base' de nuestra aplicación, se definieron los demás componentes, como son la representación de la población, la manera de seleccionar los individuos, la función *fitness*, etc., y así parametrizar el algoritmo para resolver el Problema del Viajante.

```
Algorithm GA:

input:
C = Coordenadas,
NumCromosomas= numero de cromosomas de la población,
MaxGen = máximo numero de generaciones,
pc = probabilidad de cruce, pm = probabilidad de mutación,
output : partición V1 U V2 = V
var
BestSoFar;   //Inidividuo con mejor fitness
InitPopulation;   //Población inicial
Population ;   //Población
BestPopulation;   //Mejor de la población
begin
        DistaceCities(Coordenadas); // Obtiene las distancias entre las ciudades,
        CreateInitPopulation(InitPopulation); // Creación de la población inicial
        EvaluatePopulation(InitPopulation);  // Evalua la población
        BestSoFar :=  individuo que tiene un mejor fitness;
        BestPopulation = BestSoFar,
        For Gen = 1 to MaxGen // Crear nueva población
        i := 0;
            While i <NCromosomas
                    Selection (Population, Individual1);
                    R := uniformly distributed random number in [0,1];
                     if r <= pc
                            Se elige otro individuo de la población;
                            Aplicar cruce a los individuos seleccionados;
                            Insertar los nuevos individuos creados en la nueva población;
                             i := i + 2;
                    else
                            Insertar Individual1 en la nueva población;
                             i := i + 1;
                    endif
                endWhile
              if i < NCromosomas
                     Selection (Population, AdditionalInd);
                     Insertar AdditionalInd  en la nueva  población;
              Endif
                Aplicar mutacion a la nueva poblacion con probabilidad Pm ;
                EvaluatePopulation (Population);
                BestSoFar := individuo con mejor fitness;
                BestPopulation  = BestSoFar ,
                InitPopulation:= NewPopulation,
         endfor
return (BestSoFar);
end
```

Figura 3.1: Propuesta el algoritmo genético implementado

El Algoritmo Genético implementado, cuyo pseudocódigo se muestra en la figura 3.1, recibe unos datos de entrada que son: la probabilidad de cruce, la probabilidad de mutación, el número máximo de generaciones, el número de individuos de nuestra población

(cromosomas) y un conjunto de coordenadas. Este conjunto de coordenadas que introducimos, son las correspondientes a las ciudades en el espacio.

Una vez obtenidos los datos de entrada comienza el programa principal. Se crean nuevas poblaciones durante un número de generaciones preestablecidas (*MaxGen*), para ir obteniendo el mejor de cada una de ellas, es decir, el camino más corto. Para ello, se utilizan los operadores típicos de los Algoritmos Genéticos (*cruce y mutación*) para crear las sucesivas poblaciones. Una vez se tengan los mejores de cada iteración obtenemos el mejor de los mejores.

Siguiendo el programa principal, en primer lugar se hallan las distancias entre las ciudades (*DistaceCities*), posteriormente se genera la población inicial (*CreateInitPopulation*) y se evalúa la población inicial generada (*EvaluatePopulation*) para finalmente hallar el mejor (*BestSoFar*). A continuación, se procede a generar la nueva población, para lo que se utiliza el bucle ***for*** con un ***while*** anidado.

El bucle ***for*** controla el número de generaciones que se van a crear, que irán de *i* al *MaxGen*. Para cada una de las iteraciones del bucle ***for***, se recorre toda la población, utilizando el bucle ***while*** que, como máximo, llegará a *Ncromosomas.* Con esto se garantiza la creación de poblaciones del mismo tamaño que la inicial.

Los pasos que se siguen en el bucle ***while*** para cada individuo de la población son los siguientes***:***

- ✓ Se selecciona un individuo (*selection*)
- ✓ Se genera un número aleatorio
- ✓ Si (***if***) ese número es menor o igual que la probabilidad de cruce entonces (***then)*** se selecciona otro individuo (*selection*), se cruzan y se introducen en la nueva población los dos hijos. Si no (***else***), se introduce directamente ese individuo en la población.

Una vez terminado el bucle ***while*** nos encontramos con otra condición, que introduce un individuo adicional por si la nueva población es menor que la original (2º ***if***). Posteriormente se utiliza el operador genético de la mutación, volviendo a evaluar la población resultante (*EvaluatePopulation*) e introduciendo el mejor en la población de mejores (*BestPopulation=BestSoFar*).

Por último, la anterior población es sustituida por la nueva para pasar a la siguiente generación (***end for***).
El resto del capítulo se dedica al diseño de los componentes más importantes del GA.

3.1.1. Representación cromosómica

Existen múltiples formas de representar los individuos en un Algoritmo Genético. Después de haber estudiado diferentes algoritmos genéticos y el problema del viajante (TSP), se elige la representación con números naturales, por creer que es la más natural para representar las ciudades. De este modo los posibles caminos se representan con vectores de ciudades.

Por lo tanto, la población se representa como una matriz de números naturales, donde cada fila representa un posible camino, el total de las filas es el número de individuos que forma la población (cromosomas) y el número de columnas es el número de genes de cada cromosoma.

$$Ciudades\ (1 \text{ x } N) = \{1, 2, 3, 4, 5, 6\}$$

N = nº de ciudades = nº genes
M = nº de cromosomas

$$Población\ (M \text{ x } N) = \begin{pmatrix} 1 & 2 & 3 & 5 & 4 & 6 \\ 1 & 5 & 4 & 3 & 2 & 6 \\ 1 & 6 & 2 & 5 & 4 & 3 \\ 1 & 3 & 2 & 6 & 5 & 4 \\ 1 & 4 & 3 & 5 & 6 & 2 \end{pmatrix}$$

Figura 3.2: Representación de una población

3.1.2. Función de evaluación para el algoritmo

Para definir la función de evaluación se toman para todos los cromosomas (caminos) la misma ciudad de origen. Una vez tomada la ciudad de origen recorremos todos los individuos de la población sumando para cada uno de ellos las distancias entre ciudades sucesivas, sumando también la distancia de la última ciudad a la primera, como se muestra en el siguiente pseudocódigo.

```
input:
Población,
Distancias,
Output:
Lista con la distancia total de cada individuo
Var
ListaRecorridos;    //Contiene la longitud de todos los caminos
DistanciaTotal;     //Almacena la distancia de un camino
peso;     //Distancia de una ciudad a otra
ruta;     //Contiene la ruta a la que se le calcula la distancia total
Begin
tam=size(poblacion);
N= tam (1,2);
M= tam (1,1);
Peso = 0;
for i =1:NCromosomas // nº de cromosomas
   // Se selecciona  la ruta i
   ruta=Poblacion(i, );
   for j = 1:NGenes-1  //nº de genes
         // Se calcula el peso de una ciudad a la siguiente
         // y almacenamos el peso asociado
                                    peso=Distancias(ruta(j),ruta(j+1))+peso;
   endfor
        //Se busca el peso correspondiente para volver a la ciudad origen
       // y se suma al peso que llevamos ya calculado.
            peso=Distancias(ruta(1),ruta(N))+peso
      // Se guarda la distancia total de recorrido del  individuo.
       ListaRecorridos = DistanciaTotal
endfor
Return (ListaRecorridos)
```

Figura 3.3: Pseudocódigo de la función de evaluación

La función *DistanceCitie,* crea una matriz simétrica *(NXN)* donde *N* es el número de genes, y cada posición de la matriz es la distancia entre las ciudades representadas por los índices de dicha matriz. Como puede verse en la figura 3.4.

```
input:
C = Coordenadas
Nºciudades=NGen
```

```
output : distancia entre ciudades
var
Distancias;  //Matriz que contiene las distancias entre ciudades
begin
j = 1;
for i= 1: Nºciudades
            Se selecciona  el punto que se encuentra en la posición de la ciudad i;
   while j <= Nº ciudades
        if i=j;
         // Es la distancia de una ciudad a si misma, por lo tanto se introduce  un valor Inf.
                                                Distancias [i, j]= Inf;
          j = j + 1;
       else
                        Se selecciona el punto que se encuentra en la posición de la ciudad j;
                                Se halla la distancia aplicando el Teorema de Pitágoras;
                                                Distancias [i, j]= distancia;
                                                Distancias [j, i]= distancia;
                                                        j = j + 1;
       end if
   end while
   j = i + 1;
 endfor
Return(Distancias)
```

Figura 3.4: Pseudocódigo de la función DistanceCitie

3.1.3. Función de selección

La función de selección permite seleccionar individuos de buena calidad para posteriormente cruzarse. Para ello nos será útil la función diseñada en el apartado anterior (3.1.2).

El objetivo del proceso de selección es favorecer a los individuos con un menor coste en su recorrido sin olvidar totalmente los individuos que tienen un coste más elevado. El método elegido para la resolución de nuestro problema es *roulette-wheel-selection*

Para lograr la función de selección requerida para el problema, se ha utilizado una función encontrada en la literatura llamada *MaxCut-Problem*, con la diferencia que en esta última, se favorecen a los individuos de mayor *fitness* y lo que se pretende es el caso contrario. Para ello lo que haremos será invertir las probabilidades, para favorecer caminos más cortos.

```
Input
Poblacion , Distancia_ciudades
Output
Individuo
Var
NuevoFitness;
Nuevo_total_fitness;
```

```
RelFitness;
CumulativeFitness;
Begin
Hallar la lista con los distancia de cada camino de la población;
Hallar la suma de todas las distancias halladas anteriormente;
Seleccionar la máxima distancia de la población;
for i=1:N  // N = número de individuos
     NuevoFitness(i)=(maximo-Lista_distancias(i))/100000;
end
Nuevo_total_fitness=sum(NuevoFitness);
for i=1:N
    RelFitness(i)=(NuevoFitness(i)/Nuevo_total_fitness);
End
// Hallamos los fitness acumulados
acumulado=0;
for j = 1:N   // N = número de individuos
    acumulado=RelFitness(j)+acumulado;
     CumulativeFitness(j)=acumulado;
end
r=rand;
 i=1;
 while (CumulativeFitness(i)<r)
    i=i+1;
 end
 Return( Individuo)
```

Figura 3.5: Pseudocódigo de la función de selección

3.1.4. Descripción de los operadores genéticos

Esta sección se centra en describir el diseño de los operadores genéticos: *el cruce y la mutación*. Uno de los objetivos que se pretenden es estudiar el comportamiento de nuestro algoritmo genético utilizando diferentes operadores de cruce, y comparar estos últimos a su vez, para determinar cual de ellos es el más apropiado para resolver el Problema del Viajante. En esta sección de describen los operadores de cruce y mutación.

***Partially-Mapped Crossover* (PMX) y *Order Crossover* (OX1)**

A la hora de diseñar estos cruces vemos dos partes diferenciadas, por un lado la obtención de los patrones que sería igual para los dos cruces, y por el otro la creación de sus respectivos descendientes. Por lo tanto, al diseñar los operadores de cruce PMX y OX1, se observa que lo único que los diferencia es la forma de generar sus descendientes una vez que se han introducido los patrones elegidos.

```
input:
Padre 1
Padre 2
```

```
Output:
Patron1
Patron2
Var
PosicionInicio  //Indica primera posición del patron;
PosicionFin;  //Indica última posición del patrón;
Begin
// Generar posiciones aleatorias para indicar primera posición y     // última posición del patrón
PosicionInicio = randint(1,1,[2 N]);
PosicionFin =randint (1,1,[2 N]);
   for i=1:( PosicionFin – PosicionInicio+1)
       Posiciones=PosicionInicio;
       PosicionInicio +1;
   endfor
   Seleccionar las ciudades que se encuentran entre las posiciones  anteriormente generadas y las
metemos en los patrones;
   for j = 1:n
        Patron1(1,j)=Padre1(Posiciones(j));
        Patron2(1,j)=Padre2(Posiciones(j));
         j= j + 1;
   endfor

Return(Patron1,Patron2)
```

Figura 3.6: Pseudocódigo para generar los patrones de PMY y OX1

En la figura 3.6 se explica la forma de generar los patrones para los dos cruces. Como puede verse inicialmente se generan dos números aleatorios que nos indican el principio y final del patrón. A continuación introduce en un array de posiciones las posiciones comprendidas entre ambos puntos (primer bucle ***for***). Finalmente, se introducen las ciudades de esas posiciones en su respectivo patrón (segundo bucle ***for***).

La forma en la que generarán los descendientes los cruces PMX y OX1 será la siguiente:

```
Input:
(Hijo ,Padre ,Patron1 ,Patron2 ,Posiciones)
Output:
Hijo
Var
PadreAux;
Begin
PadreAux:=Padre
for i =1:N       // N=Nºgenes
   if vacio (Posiciones = i)    // Si "i" no se encuentra entre los valores del array posiciones, mira para
ver si el padre(i) está en el hijo
            if vacio (Hijo = Padre(i))    // La ciudad no se encuentra en el descendiente
            Hijo(i) = Padre(i);
                                  else    // La ciudad se encuentra en el descendiente
             while no vacío (find (Patron2 = PadreAux(i)))  // Mientras la ciudad se encuentre en el
descendiente, elegimos otra ciudad
```

```
            posicion = find (Patron2 = PadreAux(i));
             Hijo(i) = Patron1(1, posicion);
             PadreAux(i ) = Hijo(i);
             endwhile
            endif
   endif
endfor
Return(Hijo)
```

Figura 3.7: Pseudocódigo del algoritmo para PMX

Con el bucle ***for*** se recorren las ciudades, comenzando desde la ciudad de partida. Seguidamente, nos encontramos una estructura ***if*** anidada dentro de otra. Con la primera de ellas se contrala que las posiciones de las ciudades que vamos a introducir no estén comprendidas entre el principio y fin de los patrones. Es decir, si esa posición no es del patrón intentaremos introducir el gen de esa posición en el descendiente. Con el ***if*** anidado se comprueba si ese gen está ya introducido en el descendiente. En caso de no estar, se introduce en el descendiente en la misma posición que se encuentra en su progenitor (rama ***then***). En caso contrario (rama ***else***), se utiliza un bucle ***while*** para buscar su posición. (Véase el ejemplo incluido en la Sección 1.3.4.1).

En el caso del algoritmo para OX1, la primera estructura ***if*** nos permite comenzar a introducir ciudades en el descendiente desde el principio, si la última posición del patrón coincide con la última posición del descendiente (***then***), o posicionarnos a partir del patrón en el descendiente para introducir las ciudades del padre (***else***).

Dentro de la estructura ***while*****,** nos encontramos tres estructuras ***if***. La primera de ellas será para introducir la ciudad en el descendiente en caso de que en ésta no se encontrara, y las otras dos para modificar los índices de búsqueda en caso de que llegaran al final del descendiente (Véase el ejemplo incluido en la Sección 1.3.4.2).

```
Input:
Hijo,Padre,PrincipioPatron,FinalPatron
Output:
Hijo
Begin
// Introducimos la ciudad origen
hijo(1,1)=padre(1,1);
if FinalPatron = N
```

```
then
  pos_padre=posicion de la ciudad inicial;
  pos_hijo= primera posición del Hijo;
else
  pos_padre=posicion de la ciudad inicial;
  pos_hijo= posición en la que comenzamos a introducir en el Hijo;
endif

    while vacio(hijo(poshijo)) //mientras vacio
            if  vacio(Hijo = Padre(pos_padre)) // La ciudad no se encuentra en el  descendiente
          then
            Introducimos la ciudad en el descendiente (Hijo);
            pos_padre=pos_padre+1;
            pos_hijo=pos_hijo + 1;
           else
            pos_padre = pos_padre+1;
          endif

        if pos_padre = N+1
        then
          pos_padre=2;
         endif

         if    pos_hijo = N+1
           then
            pos_ hijo = 2;
        endif
    endwhile
end
Return(Hijo)
```

Figura 3.8: Pseudocódigo del cruce OX1

***Order Based Crossover* (OX2)**

En este cruce el patrón se genera de diferente manera a los dos cruces anteriores, simplemente se seleccionan 'x' posiciones en un padre.

Para generar el hijo, como comentamos en la Sección 1.3.4.3, se impone el orden en el que se encuentran esas ciudades en las posiciones seleccionadas de un progenitor en el descendiente de otro.

```
Input:
Hijo, Padre, Patron
Output:
Hijo
Var
Posiciones;
Begin
k=1;
for i=1:S            // S = nº elementos del patrón
//Introduce en una array auxiliar las ciudades del padre situadas en las //posiciones indicadas en el
patrón
   aux(i) = Padre(Patron(i));
endfor
k = 1;
for j=1:N            // N =  nº genes
   if Hijo (j) se encuentra en aux
          //Introducimos un array la posición "j"
     Posiciones (k) = j;
     k=k+1;
   endif
endfor

for k=1:S
//Introducimos en el hijo las ciudades en el orden en que se encuentran en el padre
   hijo(posiciones(k))=aux(k);
endfor
end
Return (Hijo)
```

Figura 3.9: Pseudocódigo para OX2

Como se observa en la figura 3.9, se utilizan tres bucles ***for.*** El primero de ellos para seleccionar las ciudades en el progenitor, el segundo para buscar la posición de esas ciudades en el descendiente, y en el tercero para imponer el orden de esas ciudades en el padre en el descendiente del otro.

***Position Based Crossover* (POS)**

El cruce POS, genera un descendiente introduciendo unas ciudades aleatorias de un progenitor y el resto de las ciudades en el orden en que aparecen en el otro. La obtención del patrón se hace de la misma forma que para el cruce OX2.

```
Input:
Hijo, Padre1, Padre2, Patron
Output:
Hijo
Var
M;              // Ngenes
N;              //Numero de posiciones que tiene el patron
Begin
//Introducimos la ciudad origen
Hijo(1)=Padre1(1,1);
//Introducimos en el hijo las ciudades seleccionadas (no lo indica patron)
for i=1:N
   Hijo(Patron(i))=Padre2(Patron(i));
endfor;
//Cogemos las posiciones del descendiente que estan vacías
//Introducimos las ciudades del otro padre
k=1;
for j=1:M  //nº ge genes
   if no se encuentra la ciudad en el Hijo
      Hijo(vacias(k))=Padre1(j);
      k=k+1;
   end if
endfor;
end
Return(Hijo)
```

Figura 3.10: Pseudocódigo del algoritmo para POS

En el pseudocódigo anterior, simplemente se utilizan dos bucles ***for*,** uno de ellos para introducir en las posiciones indicadas por el patrón las ciudades que se encuentran en esas mismas posiciones en el progenitor (primer bucle ***for***) y el otro para introducir las ciudades del otro progenitor (segundo ***for***).

***Alternating-Position Crossover* (AP)**

```
Input:
Padre1,Padre2
Output:
Hijo
Var
HijoAux;
Begin
% N=NGen
% M=NCromos
tam=size(padre1);
//Creamos un hijo auxiliar con el doble de posiciones, todas ellas a cero.
      k=1;
      for m=1:N  // nº de genes
//Introducimos las ciudades de los dos padre intercaladas en el hijo auxiliar
         HijoAux(k)=Padre1(m);
         HijoAux(k+1)=Padre2(m);
         k=(m*2)+1;
```

```
      endfor;
      k=1;
      m=1;

    while (m<=N)
       if No se encuentra(Hijo = HijoAux(k))
I                                          //Introducimos la ciudad
          Hijo(m)=HijoAux(k);
          m=m+1;
          k=k+1;
       else
                                 Cogemos la siguiente ciudad del HijoAux;
          k=k+1;
       endif
    endWhile
    Return (Hijo)
```

Figura 3.11: Pseudocódigo del algoritmo para AP

Como nos indica el propio nombre del cruce (*Alternating-Position*) creamos el hijo alternando las ciudades del descendiente. Para ello, se utiliza una estructura ***for***, que genera un individuo con el doble del número de genes. En este individuo se introducen todos lo genes alternados.

La estructura ***while*** permite crear el individuo final con tamaño igual al número de genes, sin ciudades repetidas. Para ello se utiliza la estructura ***if*** anidada, que en el caso de encontrarse una ciudad ya incluida, pasa a la siguiente ciudad en el hijo auxiliar (Rama ***else***).

Mutación

A la hora de realizar el estudio experimental con los diferentes operadores de cruce, se decidió estudiarlo con el mismo operador de mutación.

La forma en la que se realiza la mutación es la siguiente:

- ✓ Se genera un umbral de mutación que será $1/N$ donde N es igual al número de genes de los individuos.
- ✓ Se recorre la población y se va comprobando cada uno de los genes (ciudades) que componen la población.
- ✓ También se genera una población de igual tamaño que la utilizada con las probabilidades de mutación de cada gen.
- ✓ Si la probabilidad de mutación de una ciudad es menor que el umbral 1/N, entonces esa ciudad muta de la siguiente manera:
    - ✓ Se genera una ciudad aleatoria.

- ✓ Se intercambian estas ciudades en el individuo, es decir, se introduce la ciudad aleatoria en la ciudad que muta y viceversa.

A continuación se muestra un pseudocódigo del mismo:

```
Input:
Poblacion
Output:
Poblacion
Var
N;   //NGen
M;  // NCromos
umbral;
Begin
umbral=1/48;
//Generamos aleatoriamente la probabilidad asociada a cada elemento de la //población
alea=rand(size(poblacion));
for i= 1: M                    // M= nº de individuos de la población
   Cogemos el individuo i;
   for j = 2:N
      if alea(i,j)<umbral
                                    Generamos una ciudad aleatoria de 1-Ngen
        //La ciudad generada aleatoriamente no puede ser igual que la ciudad origen
         while CiudadAleatoria==CiudadOrigen
              Generamos otra ciudad aleatoria
         endwhile
          Guardamos la posicion de la ciudad que se genera aleatoriamente;
            Sustituimos la ciudad que muta por la hayada aleatoriamente;
          Individuo(j)=CiudadMutacion;
          Cambiamos la ciudad aleatoria por la que muta
         endif
      endfor
  //Guardamos en la poblacion el camino mutado
Población():= Individuo;
 endfor
end
Return (población)
```

Figura 3.12: Pseudocódigo para implementar la mutación

3.2. Descripción informática

En esta sección se presenta la descripción informática del proyecto. En ella se describen los requisitos, las tareas, las estructuras de datos y la relación de funciones implementadas.

3.2.1. Herramientas utilizadas

MATLAB es una poderosa herramienta para la investigación, el cálculo, la simulación y el modelado de datos. Es un entorno de desarrollo para llevar a cabo cálculos matemáticos y la visualización gráfica de los mismos.
MATLAB es el nombre abreviado de "MATrix LABoratory" y fue originariamente escrito para proveer acceso fácil al software matricial desarrollado por los proyectos LINPACK y EISPACK. Este producto proporciona un medio para resolver problemas complejos.

La elección de MATLAB para la implementación de los algoritmos probados se debe principalmente a las numerosas ventajas que presenta respecto al resto de lenguajes de programación. Las principales razones para su elección fueron, que dispone de una herramienta muy eficiente de depuración, y que el elemento básico de trabajo son las matrices. Además se ha valorado la extraordinaria ayuda que posee para encontrar la información que se desea y su capacidad de realizar una amplia variedad de gráficos en dos y tres dimensiones.

MARLAB dispone también un amplio conjunto de librerías de programas de apoyo especializados, denominados *ToolBoxes.* Estos *ToolBoxes* cubren prácticamente casi todas las áreas principales en el mundo de la ingeniería. Destacan el *toolbox* de procesamiento de imágenes, redes neuronales, estadística, matemáticas simbólicas etc.

En la actualidad, MATLAB tiene un alto nivel de implantación en el mundo universitario, así como en departamentos de investigación de muchas empresas. Los usos más característicos los encontramos en áreas de computación y cálculo numérico tradicional, estadística, análisis de series temporales para el proceso digital de imágenes etc.

En la realización de nuestra aplicación hemos empleado el compilador MATLAB en su versión 6.1.

3.2.2. Especificación de requisitos

En la siguiente tabla se describen los requisitos básicos del presente proyecto.

Requisitos	Descripción
RF 1	La aplicación pide al usuario que introduzca el número de cromosomas que quiere para su población. Valores de 50 a 1000.
RF 1.1	La aplicación pide al usuario que introduzca el número de generaciones. Valores de 10 a 1000.
RF 1.2	La aplicación pide que se elija el cruce.
RF 2	La aplicación lee el fichero con las coordenadas de nuestras ciudades.
RF 3	Ejecuta el algoritmo elegido ofreciendo el resultado en pantalla y en una estructura de memoria
RF 3.1	El resultado que ofrece el programa es: el tiempo que tarda el algoritmo, la menor distancia recorrida, la lista de todas las distancias, y el mejor camino.
RF 4	La aplicación guarda todo el espacio de trabajo ofrecido por MATLAB en disco.

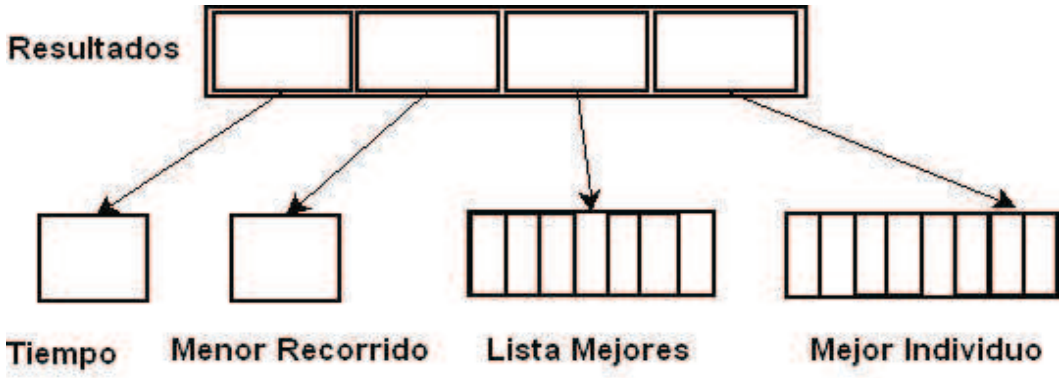
3.2.3. Análisis: Tareas

A continuación se describen las tareas que componen la aplicación.

Tarea	Descripción	Requisitos Asociados
Tarea 1: Entrada de opciones	La aplicación pide al usuario que introduzca los siguientes datos: número de cromosomas, número de generaciones y el cruce.	1, 1.1,1.2
Tarea 2: Leer fichero	En esta tarea, la aplicación se ocupa de leer el fichero introducido que guardará una estructura en memoria.	2
Tarea 3: Ejecutar AG	Esta tarea consiste en ejecutar el algoritmo genético. Los resultados que se piden son: el tiempo que tarda el algoritmo, la menor distancia recorrida, la lista de todas las distancias, y el mejor camino.	3, 3.1
Tarea 4: Mostrar resultados	En esta etapa, la aplicación muestra los resultados obtenidos. Los muestra por pantalla.	3.1
Tarea 5: Guardar Resultados	La aplicación guarda los resultados.	4

3.2.4. Estructura de datos

Se han seguido las recomendaciones de MATLAB respecto a que son más rápidas sus funciones vectoriales que sus contrapartidas escalares. Para aprovechar el gran potencial de esta aplicación, las dos estructuras que hemos utilizado son: las matrices y los vectores. A diferencia de otros lenguajes de programación no necesitamos definir previamente la estructura de datos.



3.2.5. Descripción de funciones

Al trabajar con MATLAB, los elementos constitutivos de vectores y matrices son números reales. Por lo tanto, los datos que hemos pasado en nuestras funciones son los anteriormente citados.

Nombre:	Descripción:
Estudioexperimental.m	Función que realiza el interfaz entre el usuario y el algoritmo genético.
AG.m	Algoritmo genético en sí. Recibe como entrada el número de cromosomas, el número de generaciones y el cruce. Devuelve el tiempo, el mejor camino, la lista de resultados y el menor coste.
DistanceCities.m	Dado el número de genes y las coordenadas nos devuelve una matriz simétrica. Donde se encuentran las distancias entre ciudades.
CreateInitPopulation.m	Recibe como datos de entrada el número de cromosomas y el número de genes. Devuelve la población inicial (matriz) de individuos con la que vamos a trabajar.
Selection.m	Esta función recibe como entrada la matriz que contiene la población y la matriz que contiene la distancia entre las ciudades y nos devuelve un individuo.
CrucePMX.m	Recibe como entrada dos individuos y devuelve sus dos descendientes después de cruzarlos.
CruceOX1.m	Igual que PMX

CruceOX2.m	Igual que PMX
CrucePOS.m	Igual que PMX
CruceAP.m	Recibe como entrada dos individuos y devuelve un descendiente.
Intercambio.m	Esta función recibe como datos de entrada el patrón con las ciudades que vamos a cambiar, el principio y el final del patrón y el número de genes. Nos devuelve como datos un vector que contiene esas ciudades entre las posiciones indicadas por el principio y final del patrón.
BestPopulation.m	Recibe como datos de entrada dos matrices que contienen las distancias entre las ciudades y la población, respectivamente. Nos devuelve un vector con el mejor de la población.
Mutacion.m	Recibe como dato de entrada una matriz con la población y devuelve como dato de salida la matriz de la población ya mutada.
Fitness.m	Recibe como datos de entrada dos matrices: una con la población y otra con las distancias entre las ciudades. Nos devuelve un vector que contiene las distancias para recorrer cada individuo.
Sacarpatron.m	Esta función recibe como datos de entrada un individuo de la población y un vector con posiciones. Nos devuelve un patrón que contiene las ciudades del individuo en esas posiciones.
Hijosvalidos.m	Esta función recibe como datos de entrada un hijo, un padre, los patrones y las posiciones asociadas a los patrones. Nos devuelve el hijo con las ciudades ya intercambiadas, usado para PMX.
Hijosvalidos1.m	Esta función recibe como datos de entrada el padre, principio y final del patrón y el número de genes. Nos devuelve el número ya cruzado (OX1).
Hijosvalidos2.m	Recibe como datos de entrada el descendiente, el progenitor y el patrón, nos devuelve el progenitor con las ciudades ya cambiadas(OX2)
Hijosvalidos3.m	Recibe como datos de entrada un hijo, los dos padres y el patrón. Nos devuelve el descendiente cruzado (POS).
Patron.m	Esta función recibe como entrada el número de genes y nos devuelve un vector con el patrón. Utilizado para PMX y OX1.
Patron2.m	Esta función recibe como entrada el número de genes y nos devuelve un vector con el patrón. Utilizado para OX2 y POS.
InsertarxMejores.m	Esta función introduce en la población el número *x* de mejores. La utilizamos para pruebas enriqueciendo la población (elitismo).

Capítulo 4

Resultados

En esta sección se presentan los resultados de nuestro estudio experimental. Se han utilizado los siguientes valores para los parámetros:

- ✓ Número de ciudades igual a 48.
- ✓ Número de individuos igual a 50, 100, 200 y 500.
- ✓ Probabilidad de cruce 0.7.
- ✓ Probabilidad de mutación: umbral igual a 1/N. Donde N=48.
- ✓ Coordenadas de nuestras ciudades tomadas desde el fichero 'coordenadas.xls'.
- ✓ Procesador AMD Athlon, 1.47 GHz y 256 MB de RAM

Una vez estudiado nuestro algoritmo con estos parámetros, se ha incluido otra variante. Se han realizado pruebas con elitismo, es decir enriqueciendo la población con *x* mejores. En nuestro caso hemos realizado pruebas con el 20% y el 30%.

Una vez detallados los parámetros, a continuación se muestra la representación de las ciudades y una de las posibles soluciones. Véanse las siguientes figuras 4.1 y 4.2

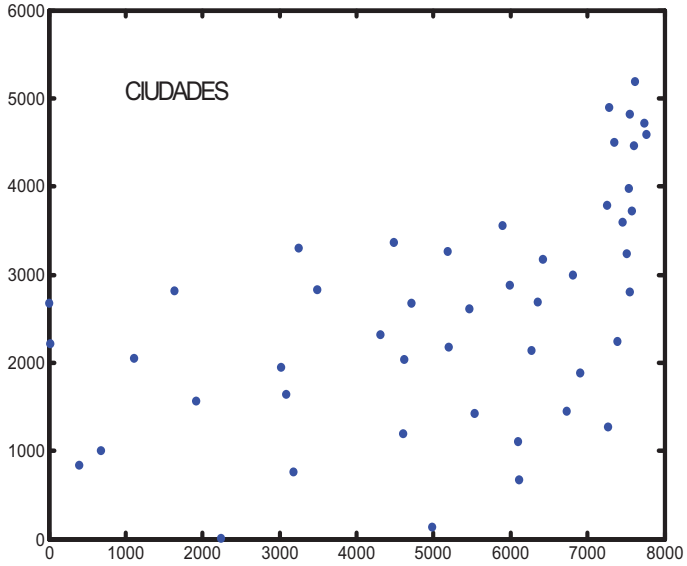


Figura 4.1: Representación de las ciudades utilizadas

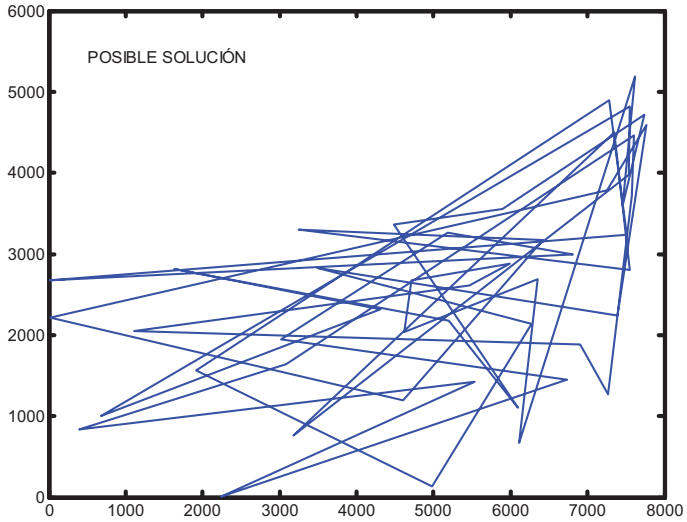


Figura 4.2: Representación de una posible solución

Las siguientes tablas nos presentan los resultados obtenidos con diferente número de individuos y diferente número de generaciones:

- Para 50 individuos

MEJOR TIEMPO	50 gener	100 gener	200 gener	500 gener
Pmx	1,0852 e+005	9,4389 e+004	9,2921 e+004	8,1709 e+004
	26,6090 seg	53,6150 seg	106,407 seg	264,0310 seg
Ox1	1,1137 e+005	9,7651 e+004	1,0360 e+005	9,2888 e+004
	26,4850 seg	52,2350 seg	105,0310 seg	261,2180 seg
Ox2	1,1019 e+005	9,6252 e+004	9,2994 e+004	8,659 e+004
	37,3910 seg	80,0630 seg	149,7340 seg	375,8280 seg
Pos	1,1237 e+005	8,4977 e+004	9,0248 e+004	8,9589 e+004
	37,8590 seg	77,9060 seg	149,00 seg	376,3440 seg
Ap	1,1583 e+005	1,1071 e+005	1,0683 e+005	9,3656 e+004
	40,8130 seg	81,2030 seg	162,1250 seg	409,3440 seg

Tabla 4.1: Resultados obtenidos para los algoritmos PMX, OX1, OX2, POS y AP con 50, 100, 200 y 500 generaciones sin elitismo

MEJOR TIEMPO	50 gener	100 gener	200 gener	500 gener
Pmx	8,8367 e+004	6,8618 e+004	7,0216 e+004	6,8640 e+004
	22,4530 seg	44,2970 seg	88,8600 seg	222,5000 seg
Ox1	9,7708 e+004	8,3576 e+004	9,0044 e+004	8,2716 e+004
	22,4370 seg	44,4220 seg	88,4060 seg	222,0620 seg
Ox2	9,4466 e+004	8,7315 e+004	7,5361 e+004	7,5844 e+004
	30,7810 seg	63,2660 seg	124,1560 seg	314,8750 seg
Pos	9,7789 e+004	9,4279 e+004	8,2297 e+004	7,7996 e+004
	30,9380 seg	62,8290 seg	126,2030 seg	312,4370 seg
Ap	9,5396 e+004	9,9735 e+004	8,5451 e+004	8,1385 e+004
	34,6250 seg	69,0160 seg	137,4690 seg	344,6880 seg

Tabla 4.2: Resultados obtenidos para los algoritmos PMX, OX1, OX2, POS y AP con 50, 100, 200 y 500 generaciones y 20% de elitismo

- **Para 100 individuos**

MEJOR TIEMPO	50 gener	100 gener	200 gener	500 gener
Pmx	1,0747 e+005	1,0015 e+005	9,3138 e+004	8,5893 e+004
	97,0150 seg	193,9380 seg	386,9690 seg	969,9840 seg
Ox1	1,0077 e+005	9,9770 e+005	9,6987 e+004	9,5997 e+004
	96,4370 seg	192,00 seg	383,9850 seg	961,8750 seg
Ox2	1,0468 e+005	1,0342 e+005	9,5777 e+004	9,4912 e+004
	119,2970 seg	237,1410 seg	474,0780 seg	1,1884 e+003 seg
Pos	1,0880 e+005	9,4953 e+004	8,5180 e+004	8,3229 e+004
	118,5310 seg	237,3120 seg	472,0630 seg	1,1898 e+003 seg
Ap	1,0512 e+005	1,0163 e+005	1,0440 e+004	1,0615 e+004
	155,4840 seg	310,3440 seg	623,7810 seg	1,5532 e+003 seg

Tabla 4.3: Resultados obtenidos para los algoritmos PMX, OX1, OX2, POS y AP con 50, 100, 200 y 500 generaciones sin elitismo

MEJOR TIEMPO	50 gener	100 gener	200 gener	500 gener
Pmx	9,0137 e+004	7,2570 e+004	6,9912 e+004	6,4009 e+004
	79,8440 seg	159,7500 seg	322,1720 seg	796,0630 seg
Ox1	8,0130 e+004	8,4173 e+004	7,2894 e+004	7,9097 e+004
	79,4220 seg	159,1880 seg	317,4220 seg	792,4220 seg
Ox2	9,3757 e+004	9,0922 e+004	7,4886 e+004	6,5522 e+004
	97,6560 seg	196,1710 seg	388,6410 seg	974,0160 seg
Pos	9,3171 e+004	8,3351 e+004	7,3325 e+004	7,0163 e+004
	97,8440 seg	195,0310 seg	390,7660 seg	976,1720 seg
Ap	8,9161 e+004	9,9072 e+004	8,2686 e+004	8,1810 e+004
	128,2970 seg	256,7660 seg	510,4850 seg	1,2812 e+003 seg

Tabla 4.4: Resultados obtenidos para los algoritmos PMX, OX1, OX2, POS y AP con 50, 100, 200 y 500 generaciones y 20% de elitismo

MEJOR TIEMPO	50 gener	100 gener	200 gener	500 gener
Pmx	9,0714 e+004	8,5775 e+004	7,1199 e+004	6,3233 e+004
	74,6250 seg	171,6250 seg	286,0940 seg	706,5940 seg
Ox1	8,4136 e+004	7,9099 e+004	8,0911 e+004	7,7927 e+004
	70,1560 seg	139,4690 seg	280,2190 seg	705,1560 seg
Ox2	1,1014 e+005	8,3348 e+004	8,2659 e+004	7,8777 e+004
	85,8900 seg	171,00 seg	341,4060 seg	861,7340 seg
Pos	1,0321 e+005	9,7238 e+004	8,0719 e+004	7,5623 e+004
	85,4840 seg	171,7820 seg	343,9380 seg	864,1250 seg
Ap	8,8927 e+004	8,8166 e+004	9,0471 e+004	9,0988 e+004
	112,1550 seg	224,2030 seg	449,6560	1,1260 e+003

Tabla 4.5: Resultados obtenidos para los algoritmos PMX, OX1, OX2, POS y AP con 50, 100, 200 y 500 generaciones y 30% de elitismo

- Para 200 individuos

MEJOR TIEMPO	50 gener	100 gener	200 gener	500 gener
Pmx	9,5619 e+004	1,0222 e+005	8,7304 e+004	9,2260 e+004
	170,0620 seg	746,2500 seg	1,4890 e+003 seg	1,4904 e+003 seg
Ox1	1,0661 e+005	1,0290 e+005	9,8434 e+004	9,6275 e+004
	172,4060 seg	741,8750 seg	1,4853 e+003 seg	3,7085 e+003 seg
Ox2	1,1061 e+005	9,9125 e+004	8,9522 e+004	8,8925 e+004
	414,3280 seg	830,9690 seg	1,6617 e+003 seg	4,1591 e+003 seg
Pos	1,0321 e+005	1,0043 e+005	8,6647 e+004	8,8738 e+004
	413,0780 seg	827,8910 seg	1,6691 e+003 seg	4,1591 e+003 seg
Ap	1,1198 e+005	9,9833 e+004	9,8377 e+004	1,0153 e+005
	612,4530 seg	1,2297 e+003 seg	2,4738 e+003 seg	6,1477 e+003 seg

Tabla 4.6: Resultados obtenidos para los algoritmos PMX, OX1, OX2, POS y AP con 50, 100, 200 y 500 generaciones sin elitismo

MEJOR TIEMPO	50 gener	100 gener	200 gener	500 gener
Pmx	8,3155 e+004	7,7342 e+004	6,4452 e+004	6,6803 e+004
	305,7030 seg	607,0780 seg	1,2084 e+003 seg	3,0155 e+003 seg
Ox1	8,7675 e+004	8,2271 e+004	7,3019 e+004	7,5477 e+004
	302,1090 seg	613,3130 seg	1,2059 e+003 seg	3,0147 e+003 seg
Ox2	9,3881 e+004	8,5913 e+004	6,9756 e+004	7,2979 e+004
	338,0160 seg	677,5470 seg	1,3532 e+003 seg	3,3768 e+003 seg
Pos	8,7840 e+004	9,2887 e+004	8,5313 e+004	6,5133 e+004
	338,7500 seg	677,3440 seg	1,3521 e+003 seg	3,3760 e+003 seg
Ap	9,3550 e+004	9,2838 e+004	8,7174 e+004	8,6260 e+004
	502,0310 seg	995,8900 seg	1,9966 e+003 seg	4,9940 e+003 seg

Tabla 4.7: Resultados obtenidos para los algoritmos PMX, OX1, OX2, POS y AP con 50, 100, 200 y 500 generaciones y 20% de elitismo

MEJOR TIEMPO	50 gener	100 gener	200 gener	500 gener
Pmx	8,9372 e+004	8,0929 e+004	6,7440 e+004	6,6948 e+004
	265,5310 seg	532,3600 seg	1,0665 e+003 seg	2,6641 e+003 seg
Ox1	8,9113 e+004	8,4242 e+004	8,3004 e+004	7,8058 e+004
	273,3120 seg	529,7190 seg	1,0627 e+003 seg	2,6505 e+003 seg
Ox2	9,7750 e+004	8,7556 e+004	7,5805 e+004	7,2156 e+004
	298,0320 seg	595,3280 seg	1188 seg	2,9635 e+003 seg
Pos	9,7947 e+004	9,1044 e+004	7,8147 e+004	7,0833 e+004
	296,6100 seg	592,9070 seg	1,1843 e+003 seg	3,0201 e+003 seg
Ap	9,4224 e+004	9,4518 e+004	8,9323 e+004	8,3391 e+004
	439,0930 seg	876,4530 seg	1,7571 e+003 seg	4,4266 e+003 seg

Tabla 4.8: Resultados obtenidos para los algoritmos PMX, OX1, OX2, POS y AP con 50, 100, 200 y 500 generaciones y 30% de elitismo

- Para 500 individuos

MEJOR TIEMPO	50 gener	100 gener	200 gener	500 gener
Pmx	1,0615 e+005	9,9296 e+004	9,7174 e+004	8,8643 e+004
	2,3145 e+003 seg	4,6173 e+004 seg	9,2319 e+003 seg	2,3297 e+004 seg
Ox1	1,0725 e+005	1,0065 e+005	9,4152 e+004	9,5597 e+004
	2,2876 e+003 seg	4,5931 e+003 seg	9,1605 e+003 seg	2,3040 e+004 seg
Ox2	1,0684 e+005	9,8961 e+004	9,6811 e+004	8,5558 e+004
	2,4026 e+003 seg	4,8595 e+003 seg	9,6300 e+003 seg	2,4098 e+004 seg
Pos	9,7664 e+004	1,0614 e+005	8,9652 e+004	8,9908 e+004
	2,4030 e+003 seg	4,8519 e+003 seg	9,6394 e+003 seg	2,4115 e+004 seg
Ap	1,0694 e+005	9,8939 e+004	1,0066 e+005	9,2984 e+004
	3,8743 e+003 seg	7,7317 e+003 seg	1,5395 e+004 seg	3,8463 e+004 seg

Tabla 4.9: Resultados obtenidos para los algoritmos PMX, OX1, OX2, POS y AP con 50, 100, 200 y 500 generaciones sin elitismo

MEJOR TIEMPO	50 gener	100 gener	200 gener	500 gener
Pmx	9,0531 e+004	7,0866 e+004	7,0395 e+004	5,7877 e+004
	1,8553 e+003 seg	3,6995 e+003 seg	7,4218 e+003 seg	1,8496 e+004 seg
Ox1	9,0839 e+004	8,0194 e+004	7,8765 e+004	7,4082 e+004
	1,8477 e+003 seg	3,6910 e+003 seg	7,3934 e+003 seg	1,8450 e+004 seg
Ox2	9,2150 e+004	8,0794 e+004	7,6262 e+004	6,6462 e+004
	1,9381 e+003 seg	3,8710 e+003 seg	7,7388 e+003 seg	1,9378 e+004 seg
Pos	8,8992 e+004	9,2149 e+004	7,2866 e+004	6,7354 e+004
	1,9381 e+003 seg	3,8690 e+003 seg	7,7431 e+003 seg	1,9334 e+004 seg
Ap	9,4047 e+004	8,8257 e+004	8,5637 e+004	7,4274 e+004
	3,1044 e+004	6,1985 e+003 seg	1,2390 e+004 seg	3,0923 e+004 seg

Tabla 4.10: Resultados obtenidos para los algoritmos PMX, OX1, OX2, POS y AP con 50, 100, 200 y 500 generaciones y 20% de elitismo

MEJOR TIEMPO	50 gener	100 gener	200 gener	500 gener
Pmx	8,0201 e+004	7,4318 e+004	7,2643 e+004	6,2983 e+004
	1,4636 e+003 seg	3,2446 e+003 seg	6,4919 e+003 seg	1,6244 e+004
Ox1	8,8131 e+004	8,5294 e+004	7,6591 e+004	7,1341 e+004
	1,4606 e+003 seg	3,2360 e+003 seg	6,4788 e+003 seg	1,6197 e+004
Ox2	9,9958 e+004	8,0754 e+004	8,4248 e+004	6,8838 e+004
	1,5394 e+003 seg	3,3976 e+003 seg	6,7952 e+003 seg	1,7001 e+004
Pos	8,8504 e+004	8,8254 e+004	8,1402 e+004	7,2677 e+004
	1,5428 e+003 seg	3,3973 e+003 seg	6,7885 e+003 seg	1,7053 e+004
Ap	9,3267 e+004	8,1092 e+004	8,4590 e+004	8,4275 e+004
	2,4498 e+003 seg	5,4462 e+003 seg	1,0855 e+004 seg	2,7514 e+004

Tabla 4.11: Resultados obtenidos para los algoritmos PMX, OX1, OX2, POS y AP con 50, 100, 200 y 500 generaciones y 30% de elitismo

4.1. Análisis de resultados

En primer lugar, analizaremos el comportamiento del algoritmo a medida que aumentamos el número de generaciones.

Como puede verse en las tablas anteriores, para las pruebas realizadas con 50, 100, 200 y 500 generaciones, observamos que el mejor camino encontrado con nuestro algoritmo, generalmente ha mejorado a medida que hemos aumentado las mismas. Por ejemplo véase los resultados para PMX, AP y OX2 con 50 individuos sin elitismo, OX1 con 100 individuos sin elitismo, OX2 para 500 individuos, POS para 200 individuos o OX2 para 500 individuos y 20% de elitismo.

En las figuras 4.3 y 4.4 se muestran dos ejemplos donde se representan las mejores soluciones obtenidas para las generaciones probadas y así observaremos la tendencia del algoritmo.

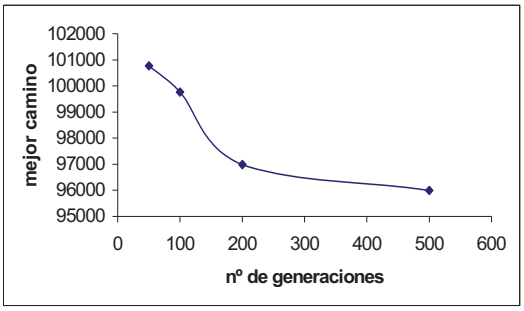


Figura 4.3: Mejores resultados para el algoritmo OX1, con 100 individuos, sin elitismo y para 50, 100, 200 y 500 generaciones

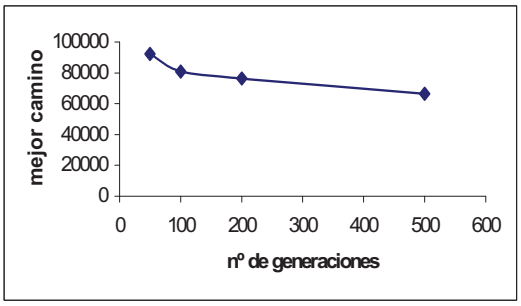


Figura 4.4: Mejores resultados para el algoritmo OX2, con 500 individuos, 20% elitismo y para 50, 100, 200 y 500 generaciones

Pero para ver la evolución del mismo más detalladamente mostraremos el comportamiento del algoritmo a medida que avanzamos en el número de generaciones. Es decir desde la generación 1 a la 50, 100, 200 y 500.

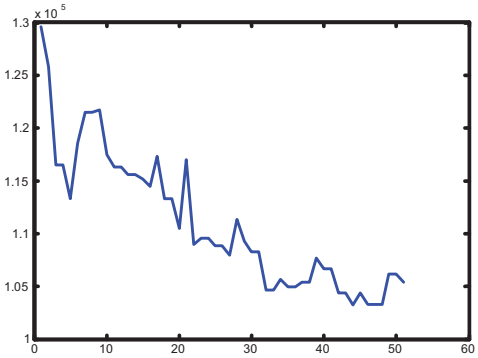


Figura 4.5: POS para 50 generaciones

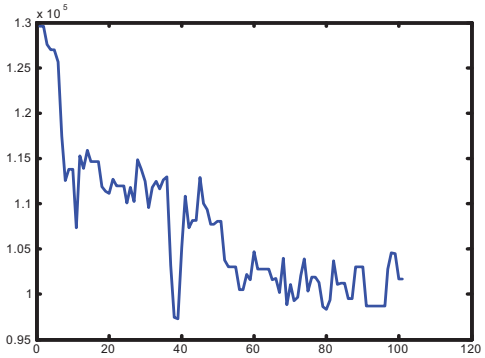


Figura 4.6: POS para 100 generaciones

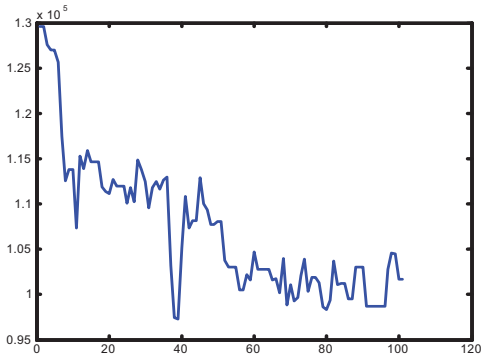


Figura 4.7: POS para 200 generaciones

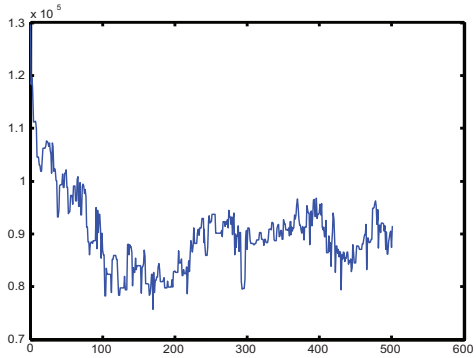


Figura 4.8: POS para 500 generaciones

Como podemos observar en los gráficos a medida que vamos aumentando el número de generaciones nuestra población va mejorando. Pero también observamos que a partir de la generación 200, aproximadamente, nuestro mejor candidato ya no mejora, incluso en el caso de 500 generaciones empeoraría. Esto se podría deber a que la población al no ser muy grande comienza a converger, es decir las nuevas poblaciones son muy parecidas y la probabilidad de mejora iría disminuyendo.

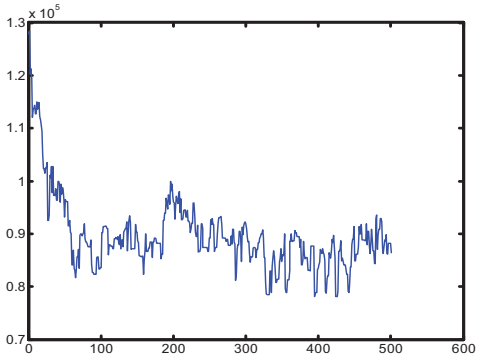


Figura 4.9: OX2 para 200 individuos y 30% de elitismo

Como puede verse claramente en la figura 4.9 la mejoría, a medida que avanzamos el número de generaciones, es muy brusca hasta la generación 70 aproximadamente. Para después no mejorar prácticamente nada. Como dijimos anteriormente nuestra población ha convergido rápidamente debido a una mala combinación entre el número de individuos, el número de generaciones, el elitismo introducido y la probabilidad que en estos algoritmos existe.

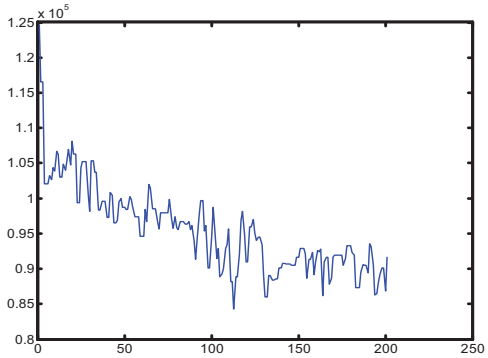


Figura 4.10: OX2 para 500 individuos y 30% de elitismo

En cambio, en la figura 4.10 observamos que en las 5 primeras generaciones desciende bruscamente, para a continuación ir descendiendo de una manera más suave, llevando una tendencia de mejora en los resultados de manera continuada y sin converger.

Una vez analizado el comportamiento de nuestro algoritmo a medida que aumentamos las generaciones, pasaremos a estudiar el comportamiento del mismo para diferentes coeficientes de mejora (elitismo). A grandes rasgos, observando las tablas de resultados, nos damos cuenta de que nuestros resultados mejoran cuando introducimos elitismo a cuando no.

De los datos obtenidos, se concluye que a la hora de introducir el 20% y el 30% de elitismo mejoran considerablemente los resultados respecto a sin elitismo. En cambio no observamos mucha diferencia del 20 al 30%, ya que no hay una diferencia muy grande en el enriquecimiento de nuestra población.

La poca diferencia existente entre el 20 y el 30 puede ser debida a que nuestros espacios de búsqueda son pequeños, ya que unos espacios muy grandes nos resultaría muy costoso de ejecutar en nuestra máquina, y además el enriquecimiento del 30% puede provocar una convergencia anticipada en nuestra búsqueda. Todo esto unido a que en una generación pueda aparecer por casualidad un individuo mejor al cruzarse dos progenitores. También puede ser debido a que el enriquecimiento del 30% resulte insuficiente para las poblaciones mayores probadas.

Otro punto a analizar, es ver cómo se comportan nuestro algoritmo aumentando la población. De manera intuitiva nos llevaría a pensar que aumentando el número de individuos aumentaría las posibilidades de encontrar una mejor solución. Para ellos hemos comparado en la siguiente tabla el número de soluciones encontradas por rangos.

Rango / Población	>100000	>90000	>80000	>70000	>60000	>50000
50	0	7	8	4	1	0
100	0	5	6	6	3	0
200	0	4	8	4	4	0
500	0	5	5	7	2	1

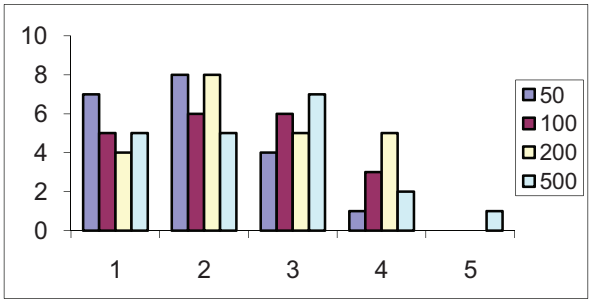


Figura 4.11: Comparativa de resultados por población

Como puede observarse, conseguimos un mayor número de individuos de alta calidad en los rangos menores a medida que aumentamos la población. Con lo que podemos suponer que tendremos una mayor probabilidad de encontrar mejores soluciones en poblaciones mayores.

Una vez estudiados estos tres puntos, pasaremos a comparar los resultados obtenidos con los diferentes cruces. Representaremos el cruce que haya obtenido un mejor resultado para cada conjunto de generaciones.

- Para 50 cromosomas

	50 gener	***100 gener***	***200 gener***	***500 gener***
Sin elitismo	PMX	POS	POS	PMX
20% elitismo	PMX	PMX	PMX	PMX

- Para 100 cromosomas

	50 gener	*100 gener*	*200 gener*	*500 gener*
Sin elitismo	OX1	POS	POS	POS
20% elitismo	OX1	PMX	PMX	PMX
30% elitismo	OX1	OX1	PMX	PMX

- Para 200 cromosomas

	50 gener	*100 gener*	*200 gener*	*500 gener*
Sin elitismo	PMX	OX2	POS	POS
20% elitismo	PMX	PMX	PMX	POS
30% elitismo	OX1	PMX	PMX	PMX

- Para 500 cromosomas

	50 gener	*100 gener*	*200 gener*	*500 gener*
Sin elitismo	POS	AP	POS	OX2
20% elitismo	POS	PMX	PMX	PMX
30% elitismo	PMX	PMX	PMX	PMX

De estas tablas se desprende que el mejor cruce con elitismo es claramente PMX. En cambio los mejores resultados obtenidos sin elitismo los obtenemos con POS,y en el último lugar se encontrarían OX2 y AP.

A continuación representaremos de manera general como es la proporción de mejores resultados obtenidos por cada uno de nuestros cruces para el algoritmo

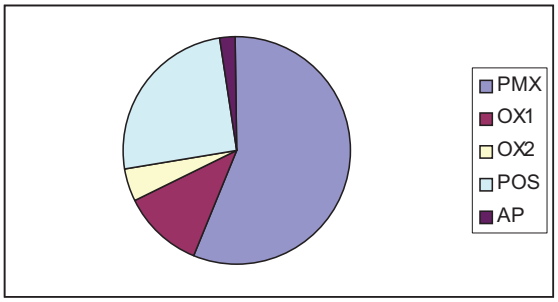


Figura 4.12: Comparativa resultados por Cruce

Y por último comentaremos los tiempos empleados en nuestras pruebas respecto al tiempo de ejecución. Como pueden verse en la tabla de resultados, los operadores de cruce más rápidos son PMX y OX1, después estarían OX2 y POS y por último AP, que es el que arroja peores resultados de tiempo en todos los experimentos.

Véanse las siguientes graficas:

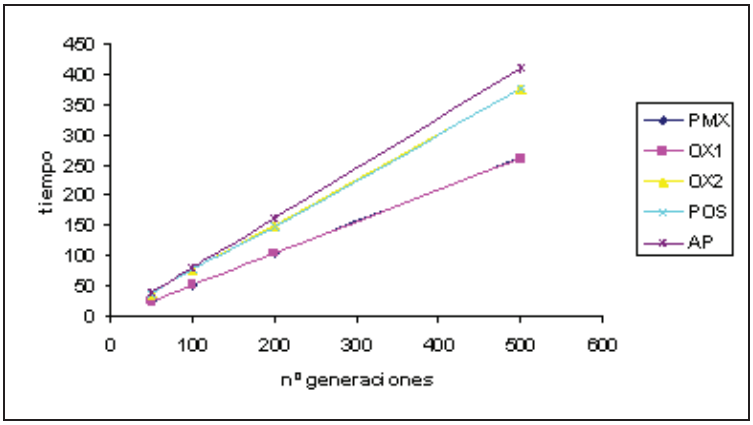


Figura 4.13: Comparativa de tiempos para los algoritmos PMX, OX1, OX2, POS y AP para 50 individuos, sin elitismo y 50,100, 200 y 500 generaciones.

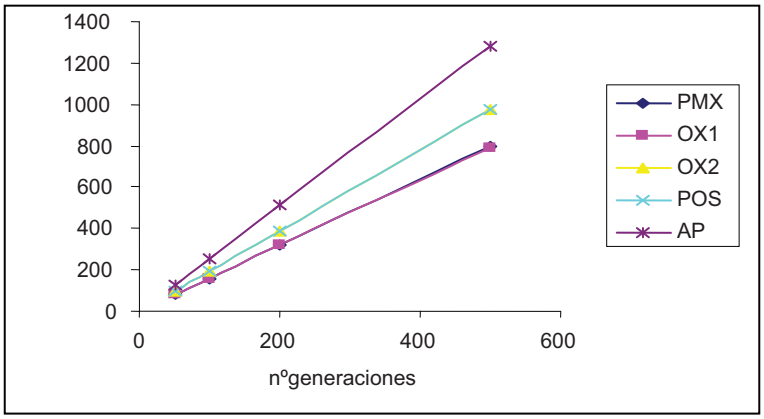


Figura 4.14: Comparativa de tiempos para los algoritmos PMX, OX1, OX2, POS y AP para 100 individuos, 20% elitismo y 50,100, 200 y 500 generaciones.

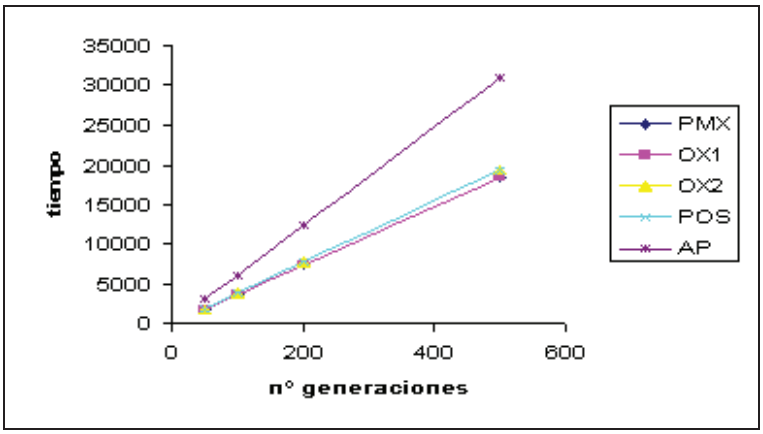


Figura 4.15: Comparativa de tiempos para los algoritmos PMX, OX1, OX2, POS y AP para 500 individuos, 20% elitismo y 50,100, 200 y 500 generaciones.

Como podemos observar en los gráficos, a medida que aumentamos las generaciones lógicamente también los tiempos. Pero también observamos que a medida que aumentamos el número de individuos los tiempos de PMX, OX1, OX2 y POS se diferencian menos, y en cambio AP aumenta su diferencia respecto a los demás. Hay que tener en cuenta que AP sólo genera un descendiente, por lo que al formar la nueva población tardará más que el resto de los cruces.

Por lo tanto el cruce que mejor se comporta a nivel de resultado y tiempo es PMX y el peor de ellos AP. Los demás cruces en cambio se comportan de una manera parecida a nivel de tiempos y de resultados, siendo POS para el que mejores resultados hemos obtenido de los tres, seguido de OX1 y OX2.

Representación de alguno de los mejores resultados:

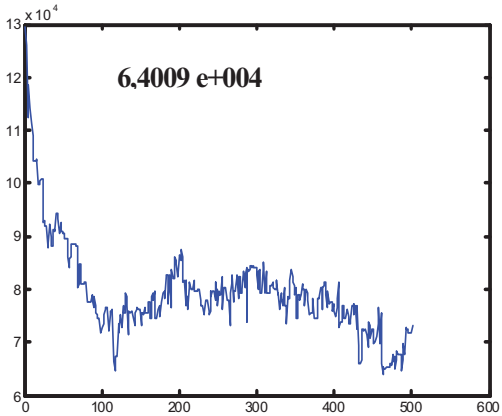


Figura 4.16: POS 200 cromosomas 500 generaciones y 20% de elitismo

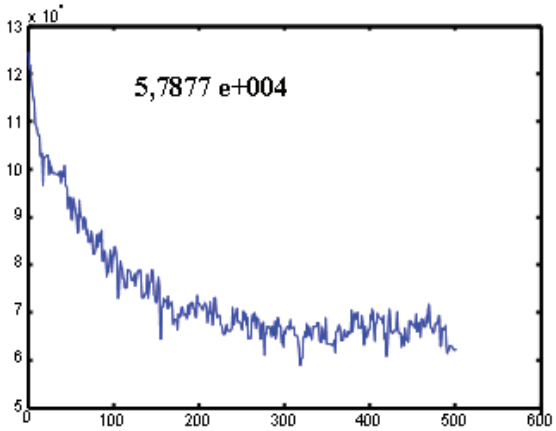


Figura 4.17: PMX 500 cromosomas 500 generaciones y 20% de elitismo

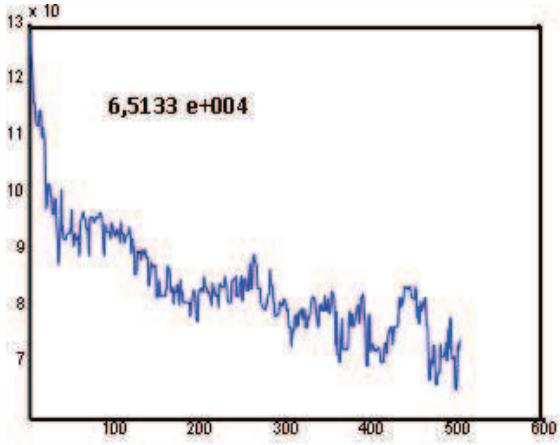


Figura 4.18: POS 200 cromosomas 500 generaciones y 20% de elitismo

Capítulo 5

Conclusiones

Una vez concluido nuestro proyecto, entre las múltiples conclusiones que podríamos sacar, se encuentra que este tipo de algoritmos (AG) nos permiten una aproximación a los resultados para problemas muy complejos, como ocurre con TSP. Y podemos decir que con nuestro algoritmo genético hemos conseguido resultados bastante satisfactorios para poblaciones y un número de generaciones relativamente pequeñas.

Un estudio experimental realizado por el *Departament of Cumputer Science* de la Universidad de Heidelberg para las mismas ciudades que nosotros, nos arroja un optimo de 36826 y nuestro mejor optimo ha sido de 57877. Esto nos hace pensar que podemos seguir probando con el algoritmo implementado con diferentes parámetros para poco a poco ir mejorando nuestra mejor solución.

Otra conclusión que podremos sacar es que el secreto de estos algoritmos para que funcionen con éxito, es una correcta combinación de los parámetros como pueden ser: nº de individuos, nº de generaciones, probabilidad de cruce, probabilidad de mutación, cruces utilizados, mutaciones etc.

En nuestro estudio podemos ver que la elección del operador de cruce es muy importante, porque un operador de cruce mal seleccionado nos llevaría directamente al fracaso.

También vemos que para poblaciones muy pequeñas se corre el riesgo de una convergencia acelerada si nos excedemos en el número de generaciones y un elitismo elevado.

5.1. Lecciones aprendidas

Las lecciones aprendidas en el transcurso de la realización del proyecto han sido muchas, no solo por el aprendizaje de un nuevo lenguaje de programación, MATLAB, sino por conocer un área de la informática hasta el momento desconocida para mí.

Alguna de ellas:

- Creación de un estudio experimental y un proyecto software de ciertas dimensiones.
- Conocimiento de los Algoritmos Genéticos.
- Conocimiento del Problema del Viajante.

5.3.Trabajos futuros

Una vez finalizado el proyecto, sería posible extender la funcionalidad del mismo. Un trabajo futuro podría ser estudiar nuestro algoritmo con otros parámetros, otros criterios de mejora, por ejemplo sólo introducir el individuo si es mejor que el peor de la población. O incluso intentar con lo realizado estudiar otro tipo de problemas NP-completos.

Bibliografía

[1] Jesús M. de la Cruz, José A. López Orozco. *Introducción a los Algoritmos Genéticos.*

[2] Departamento de Ciencias de la Computación e Inteligencia Atificial. Universidad del Pais Vasco. *Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators.*

[3] Kyung-Mi, Takeshi Yamakawa, Keon-Myung Lee. *Genetic Algorithm Approaches to Job Shop Scheduling Problems: An Overview.*

[4] Chistopher R. Houck, Jeffery A. Joines, Michael G.Kay. *A Genetic Algorithm Function Optimization: A Matlab Implementation.*

[5] Oliver Dolezal, Thomas Hofmeister, Hanno Lefmann. *A Comparison of Approximation Algorithms for MAXCUT-Problem.*

[6] Pedro Larrañaga e Iñaqui Inza. Universidad del Pais Vasco. *Heurísticos en Optimización Combinatorial.*

[7] *Aprenda Matlab 5.3 como si estuviera en primero*. Escuela Superior de Ingenieros Industriales. Universidad de Navarra.

[8] Gregorio Hernández Peñalver, Facultad de Informática UPM. *Teoría de Grafos. Complejidad y Grafos.*

[9] Departament of Computer Science University of Bahrain. *A Hybrid Genetic Algorithm for task Scheduling in Multiprocessor Real-Time Systems.*

[10] L.Recalde-C.P.S. *Esquemas Algorítmicos- Algoritmos genéticos*.

[11] Gabriela Ochoa, http://www.ldc.usb.ve/gabro. *Análisis de Algoritmos, Optimización Clásica, Heurísticas.*

[12] Juan Antonio Infante, José María Rey. *Introducción a Matlab.*