UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES
A C A T L Á N

TESIS.

"APLICACIÓN DEL ALGORITMO GENÉTICO AL PROBLEMA DEL AGENTE VIAJERO."

QUE PARA OBTENER EL TITULO DE LICENCIADO EN MATEMÁTICAS APLICADAS Y COMPUTACIÓN

P R E S E N T A :

GUSTAVO NAVARRETE SANTANDER

ASESOR:
M en C SARA CAMACHO CANCINO.

NAUCALPAN, EDO DE MÉXICO, Junio 2007

F E S UNAM ACATLÁN

DEDICATORIA:

Dedico la presente tesis a los seres que más amo en el mundo:

A Dios que me dio la oportunidad de la vida.

A mis padres Pedro y Paula que me entregaron todo su cariño, su amor, sus consejos y sus vidas.

A mis hermanos Alejandra, Arturo, Daniel Juan, Carlos y Raúl por estar a mi lado y ser parte de mi.

A Diana Leslie el amor de mi vida, mi mayor felicidad y la luz de mi existir.

A la UNAM fuente de conocimiento de millones de mexicanos.

AGRADECIMIENTOS:

A Dios. Gracias por permitirme llegar a este momento.

A mis Padres Gracias por cuidarme, educarme y amarme. Gracias Papá, Gracias Mamá.

A mis hermanos Arturo y Alejandra que me dieron siempre su apoyo a lo largo de mi carrera y son mi ejemplo a seguir. Gracias Toby y Ale.

A Diana Leslie Muchas Gracias por todo mi amor, eres mi vida y la razón de ser de mi existencia. Te amo.

A mis hermanos Daniel, Carlos, Juan y Raúl. Gracias por estar conmigo a lo largo del camino.

A Cristina, Fernando, Ricardo, a todas mis sobrinas y sobrinos. Gracias por ser mi familia.

A la Universidad Nacional Autónoma de México, institución y profesores sin cuyo apoyo no podría lograr las metas de mi vida.

A la Facultad de Estudios Superiores Acatlan, que me proporciono la preparación y formación profesional, que me acompañaran por siempre.

A la Licenciada en Matemáticas Aplicadas y Computación Sara Camacho García. Por todo su apoyo y paciencia para la realización de este trabajo de Tesis.

A mis amigos Ángel, Enrique e Ismael. Gracias por su amistad.

A mis compañeros Arturo, Cesar, Edgar, Eduardo, Hugo, Omar y Roberto. Gracias por todas las alegrías a lo largo de la carrera.

Al equipo de Fútbol PUMAS de la UNAM Gracias por todos los buenos momentos.

México, PUMAS, Universidad.

INDICE

INTRODUCCIÓN

Dentro de los paradigmas de aprendizaje automático, los algoritmos genéticos se presentan como uno de los más prometedores en el camino de evolución hacia la inteligencia artificial. Normalmente se aplican en problemas de optimización, pero constantemente se expande su campo de aplicación; debido a que actualmente han demostrado su validez en múltiples áreas.

El algoritmo genético es una técnica de búsqueda basada en la teoría de la evolución de Darwin. Comúnmente se le define como un algoritmo matemático que transforma un conjunto de objetos matemáticos individuales con respecto al tiempo, usando operaciones modeladas de acuerdo al principio Darwiniano de reproducción y supervivencia del más apto, por medio de una serie de operaciones genéticas entre las que destaca la recombinación sexual. Cada uno de estos objetos matemáticos suele ser una cadena de caracteres (letras o números) de longitud fija que se ajusta al modelo de las cadenas de cromosomas, y se les asocia con una función matemática que refleja su aptitud para resolver un problema.

En el ambiente del desarrollo de software, los algoritmos genéticos poseen un gran interés por la evidente analogía que convierte al desarrollador en un creador de "vida" virtual.

Los algoritmos genéticos se pueden entender como una estrategia de búsqueda. El algoritmo selecciona una serie de elementos tratando de encontrar la combinación óptima de éstos, utilizando para ello la experiencia adquirida en anteriores combinaciones, almacenadas en patrones genéticos. Sin embargo, en este caso, las posibilidades de la metodología trasciende de la búsqueda heurística, debido a una razón de peso: el algoritmo no trabaja Top-Down sino Bottom-Up: es decir, no es necesario disponer de un conocimiento profundo del problema a resolver, para que éste pueda ser descompuesto en sub-problemas, sino que se divide en estructuras simples que interactúan, dejando que sea la evolución quien realice el trabajo. Basta con ser capaz de identificar cualitativamente en qué casos los patrones se acercan o se alejan de la solución buscada, para que el sistema se perfeccione automáticamente, desarrollando sus propios comportamientos, funcionalidades y soluciones. De esta manera el tiempo y la evolución realizan el trabajo de mejorar la especie y converger hacía la solución. . Se puede considerar que éste modo de proceder ha demostrado ya su utilidad en el ambiente de la vida biológica, transformando las especies naturales que pueblan el planeta, incluyendo al genero humano.

El problema elegido para ser resuelto mediante un algoritmo genético es el problema del agente viajero, ya que por sus características es ideal para exponer las virtudes de los algoritmos genéticos, además es el prototipo de los problemas NP-Completos, y su importancia radica en que un problema NP-Completo tiene la característica de que todo problema en NP se reduce polinomialmente a él. Por lo tanto es importante resolver problemas en NP en tiempo polinomial ya que si se

resuelve uno, se podrán resolver todos. Investigando los antecedentes del problema se encontraron trabajos como el de la tesis de Ma. Del Rosario Pilar Varela Hernández que resolvió el problema del agente viajero estudiando algoritmos de uso común dentro de la técnica de búsqueda local como son el 2-Optimal y 3-Optimal. Por otro lado, Angel Reyes González también trabajó con el PAV (problema del agente viajero), la virtud de este trabajo es que se resolvió con el método Ariadnes's Clew método innovador inventado por el Doctor Juan Manuel Ahuactzin Larios que trabaja con dos subalgoritmos, llamados Search y Explore respectivamente. El algoritmo Explore recoge información del espacio de búsqueda incrementando su resolución en cada iteración y colocando marcas en el espacio de búsqueda. Y el algoritmo Search es un método de búsqueda local que constantemente checa si el mínimo o máximo global puede ser alcanzado. El PAV fue probado para 100 y 200 ciudades. Por lo que tomando como antecedente el trabajo realizado en las tesis antes mencionadas y sabiendo que el PAV es un problema típico tanto de optimización combinatoria como un problema de decisión o bien un problema euclidiano que puede ser resuelto con técnicas exactas de búsqueda exhaustiva o con técnicas aproximadas, se decidió desarrollar y programar un algoritmo genético para un caso simétrico.

En este sentido los objetivos del presente trabajo se pueden dividir de la siguiente manera:

Crear un algoritmo genético para resolver un caso simétrico del problema del agente viajero. Situando a los algoritmos genéticos dentro del ambiente de la informática evolutiva y el problema del agente viajero como problema NP-Completo y prototipo de problema a resolver aplicando algoritmos genéticos.

Programar una aplicación de cómputo en la plataforma de desarrollo .NET que implemente el algoritmo genético creado y el algoritmo del mejor vecino como algoritmo de comparación, ligando las técnicas de desarrollo de software con los conceptos del algoritmo genético y comparando el desempeño de ambos algoritmos mediante ejecuciones del programa, analizando los resultados obtenidos por cada uno de ellos de acuerdo a la mejor solución obtenida y tiempo de ejecución.

El contenido del presente trabajo de tesis se divide en seis capítulos con la siguiente temática:

CAPÍTULO I. INFORMÁTICA EVOLUTIVA.

El objetivo del capitulo es establecer el marco teórico dentro del cuál, se desarrolla la investigación. Inicia por explicar al lector el concepto de algoritmo, algoritmo de búsqueda y los diversos procedimientos existentes para darles solución, ubicando dentro de ellos a los algoritmos genéticos. A continuación se expone un repaso histórico de la Teoría de la Evolución de las Especies, desde la

antigua Grecia hasta Darwin y Wallace; a continuación se relaciona esta teoría con los conceptos modernos de Genética que permiten explicar los procesos de selección natural. Por último se expone el proceso histórico que llevo al desarrollo de la idea de imitar la selección natural para resolver problemas mediante diferentes técnicas, entre ellas los algoritmos genéticos.

CAPÍTULO II. ALGORITMOS GENÉTICOS.

Este capítulo se enfoca en definir el concepto del algoritmo genético y las partes que lo conforman. Después de una breve reseña histórica sobre su desarrollo, se enuncia la definición formal del concepto, así como la explicación del enfoque que da un algoritmo genético a un problema. Se exponen las diferentes maneras de codificar un problema, el concepto de población, los criterios de paro, además de las técnicas de selección, evaluación de las soluciones y los operadores genéticos generadores de diversidad como la mutación y el cruce. Se exponen las ventajas y desventajas de esta técnica y cuales son los casos donde se recomienda su uso. Por último se proporcionan una serie de consejos que llevan a un mejor entendimiento y aplicación de los algoritmos genéticos.

CAPÍTULO III. EL PROBLEMA DEL AGENTE VIAJERO.

Este capítulo se refiere al "Problema del Agente Viajero", primeramente su concepto y las múltiples aplicaciones que existen de él. A continuación se estudian algunas de las técnicas que se han aplicado para solucionarlo; estas son "El procedimiento de Little", la Trayectoria Mínima de Redes y el algoritmo del Mejor Vecino. Además se comentan algunas consideraciones sobre el problema y su complejidad.

CAPÍTULO IV. DESARROLLO DEL MODELO.

Este capítulo expone el proceso de desarrollo del algoritmo genético, comenzando con el criterio que se utilizará para detener el algoritmo, la representación que se le dará al problema mediante la codificación de la población de posibles soluciones, se establece el criterio de selección de los individuos para construir la siguiente generación, además de estudiar el modo de uso de los operadores de cruce y mutación para generar diversidad y facilitar la convergencia del algoritmo.

CAPÍTULO V. DISEÑO DEL SOFTWARE.

Las implicaciones del desarrollo del programa que utiliza el algoritmo genético son el tema de este capítulo, En él se explican los criterios de elección del lenguaje de programación, se incluyen los diagramas de flujo que ilustran el trabajo del programa y la lógica de programación; así como la implementación a código por medio de textos en seudo código, además de algunas consideraciones como requerimientos de hardware y software para el uso del programa.

CAPÍTULO VI. ANÁLISIS DE RESULTADOS.

Durante este capítulo se realiza una prueba del programa con diferente número de ciudades a visitar y expone los resultados obtenidos. Para facilitar el análisis se realiza un comparativo entre el algoritmo genético y el algoritmo del Mejor Vecino, mostrando los resultados en gráficas que permiten un mejor y más rápido análisis para la extracción de conclusiones.

El presente trabajo esta dirigido a lectores interesados en el tema de los algoritmos genéticos, su implementación y el desarrollo de software de aplicación basado en ellos. En este trabajo se estudian los algoritmos genéticos desde su concepción teórica, hasta su implementación práctica y aplicación a un caso simétrico del Problema del Agente Viajero, mismo que es a su vez estudiado en sus diferentes variantes y posibles rutas de solución. Ante todo ligando los conceptos teóricos de resolución de algoritmos con las técnicas practicas de desarrollo de software representadas por el paradigma orientado a objetos, los diagramas UML, de flujo de datos, el seudo código y la codificación en el lenguaje de programación Visual C# .NET.

CAPÍTULO I

INFORMÁTICA EVOLUTIVA

Todos los seres humanos aplican algoritmos constantemente para obtener los resultados deseados, de hecho una simple receta de cocina es un algoritmo, aún cuando resulte poco parecido a un cálculo matemático o a un programa informático; el punto de comparación consiste en que en realidad ambos expresan una serie de pasos que si se siguen fielmente garantizan el logro de un resultado deseado. Los algoritmos son el tema inicial de este capítulo, se analizan sus características y se centra la exposición en los algoritmos de búsqueda y en un tipo especial de algoritmos, los algoritmos genéticos. A continuación se explica la teoría de la evolución de las especies y el como dicha teoría se complementa con los conceptos actuales de genética, además se estudian las opciones que junto a los algoritmos genéticos conforman el campo de la informática evolutiva.

1.1 Algoritmos de búsqueda.

La palabra algoritmo tiene su origen de varias traducciones y corrupciones, del nombre del matemático árabe Al-Juarismi quien escribió el primer libro sobre álgebra, en el año 825 y que sería traducido al latín en el siglo XII. El concepto de algoritmo como un procedimiento seguro y metódico para lograr un resultado ha tenido vigencia por siglos; sin embargo se haría más notable gracias a la obra matemática de pensadores como Hilbert, Church y Turing que revolucionaron las ciencias matemáticas durante el primer tercio del siglo XX. La transferencia del concepto de algoritmo de matemáticas a la informática ocurriría en gran parte gracias a ellos. Actualmente se puede equiparar "algoritmo" con el amplio grupo de objetos que cubren los programas y sistemas operativos de las computadoras y que les permiten ejecutar sus tareas.

Las características fundamentales de un algoritmo son las siguientes:

- **Neutralidad de material.** Un algoritmo puede ser realizado en papel, una pizarra, medios electrónicos, etc; pues su naturaleza es eminentemente abstracta y por lo tanto no importa el medio material en que está encarnado.
- **Comportamiento ciego o mecánico.** Aún cuando su inventor fue una mente humana inteligente, el algoritmo no requiere en su aplicación de inteligencia, sino que la ejecución exacta de los pasos prescritos debe obtener el resultado deseado.

Los algoritmos de búsqueda por su parte abarcan prácticamente todos los ámbitos, aunque en el campo de la informática habitualmente se habla de búsqueda cuando se necesita hallar cierta información dentro de un conjunto de datos almacenados y de acuerdo a un determinado criterio de búsqueda; sin embargo aquí se hace referencia a

otro tipo de búsqueda. En este caso, dado un espacio con el conjunto de las posibles soluciones de un problema y partiendo de una solución inicial, el algoritmo será capaz de encontrar la mejor o única solución.

En la mayoría de los casos la búsqueda de la solución estará guiada por una función de evaluación que indica que tan buena es ésta, su costo o lo cerca que está de la solución final. Si se conoce dicha función el problema se convierte en un problema de optimización, donde se deben encontrar los valores que maximizan la función objetivo, de evaluación, fitness o que permiten minimizar el costo. En términos formales se dice que optimizar consiste en que dada una función F_x de n variables, se encuentre la combinación de valores de X tales que $F(X_1, X_2, ..., X_n)$ = Máximo. Se puede hablar del mismo modo de minimizar en lugar de maximizar, ya que minimizar equivale a maximizar $-F_X$.

Generalmente este tipo de problemas de optimización son tratados por la rama de las matemáticas llamada Investigación de Operaciones. La Investigación de Operaciones es definida por Churchman, Ackoff y Arnold como "*La aplicación, por grupos interdisciplinarios, del método científico a problemas relacionados con el control de las organizaciones o sistemas (hombre-máquina) a fin de que se produzcan soluciones que sirvan a toda la organización*" (1)[1]. En realidad, actualmente la investigación de operaciones engloba una gran cantidad de modelos matemáticos para las más diversas situaciones. Se observa como surge esta disciplina, mediante un breve repaso de la historia de esta rama de las matemáticas ya que si bien sus inicios se reconocen en la segunda guerra mundial, tiene como antecedentes los modelos matemáticos de los economistas Quesney (1759), Walras (1874). Los modelos lineales de Jordan (1873), Minkowsky (1896) y Farkas (1903). Los modelos lineales probabilísticos de Markov a fines del siglo XIX. Los modelos de líneas de espera de Erlang (1900), los problemas de asignación con los húngaros König (1920) y Egervary (1930). Los problemas de distribución cimentados por el ruso Kantorovich (1939). La teoría de juegos iniciada por Von Neuman (1937). Esto permite hacer notar que estos modelos matemáticos precursores de la investigación de operaciones estaban basados en el Cálculo Diferencial e integral de Newton, Lagrange, Laplace, Lebesgue, Leibnitz, Reimman, Stieltjes, por mencionar algunos y la Probabilidad y Estadística de Bernoulli, Poisson, Gauss, Bayes, Gosset, Snedecor, etc; pero sería durante la segunda guerra mundial que dichas técnicas tomaron un verdadero auge. Primero en la lógica estratégica para vencer al enemigo y al finalizar la guerra en la logística de la distribución de los recursos militares de los aliados, dispersos por el mundo. Fue debido a este problema que la fuerza aérea norteamericana por medio de su centro de investigaciones Rand Corporation, comisionó a un grupo de matemáticos para dicha tarea. El doctor George Dantzig en 1947, resumiendo el trabajo de muchos de sus precursores inventó el método Simplex, con lo cual dio inicio a la programación lineal. Con el avance de las computadoras digitales se extendería la investigación de operaciones durante la década de los cincuentas con las áreas de Programación Dinámica (Bellman), Programación No

[1] Ackoff Russell, Fundamentos de la investigación de operaciones.
México, Ed Limusa. p 20

Lineal (Kuhn y Tucker), Programación Entera (Gomory), Redes de Optimización (Ford y Fulkerson), Simulación (Markowitz), Inventarios (Arrow, Karlin, Scarf, Whitin), Análisis de Decisiones (Rafia) y Procesos Markovianos de decisión (Howard).

Con toda esta variedad de disciplinas y grado de especialización se cuenta con una buena cobertura para la mayor parte de los problemas de optimización y búsqueda; dentro de dichos problemas se tiene una serie de clasificaciones de acuerdo a ciertas características, como se expone a continuación.

En muchos casos los problemas de optimización no siempre están formulados claramente. En ocasiones la función F no se conoce y debe aproximarse mediante polinomios o funciones que calculan una función aproximada a la función objetivo. En estos casos el problema se reduce a calcular coeficientes y se habla de optimización paramétrica.

En otros casos, la función de evaluación o no existe, o no es estática, sino que viene dada por el entorno de la solución. Este tipo de optimización se suele encontrar en problemas de vida artificial o el "*Dilema del prisionero*", usado para modelar interacciones sociales.

Otros casos son en los que se trata de optimizar F_c, donde C es una combinación de diferentes elementos que pueden tomar un número finito de valores; pueden ser combinaciones con o sin repetición, o incluso permutaciones; en estos casos se les denomina como problemas de optimización combinatoria.

No siempre, el espacio de búsqueda completo contiene soluciones válidas; en algunos casos, los valores de las variables se sitúan dentro de un rango, más allá del cual la solución es inválida. Se trata entonces de un problema de optimización con restricciones. En estos casos de trata de maximizar F_{xi} $(X_1, X_2, ... X_n)$ de n variables o incógnitas, llamadas función objetivo y sujeta a un conjunto m de restricciones o limitaciones expresadas por las funciones:

$$q_1 (X_1, X_2, ... X_n) <> b_1$$
$$q_2 (X_1, X_2, ... X_n) <> b_2$$
" " " " " "
$$q_m (X_1, X_2, ... X_n) <> b_m$$

en las que m (número de restricciones) puede ser mayor o menor que n (número de incógnitas).

Si la función objetivo y las restricciones del problema están formadas por expresiones de primer grado y las variables son no negativas, se tiene un caso de programación lineal. Si la función objetivo y las restricciones son no lineales se habla de programación no lineal. Se puede distinguir también los casos estáticos en los que se busca optimizar la función objetivo considerando que la decisión tomada no es afectada por decisiones previas y no afectará decisiones futuras, y el caso dinámico en el que debe tomarse una secuencia de decisiones y donde cada decisión afectará a las futuras.

Finalmente, se puede distinguir el caso de la programación determinística y probabilística. En la primera se trabaja con un modelo donde los coeficientes en la función objetivo y el conjunto de restricciones son fijos y conocidos. En programación probabilística o estocástica existen elementos de incertidumbre que pueden introducirse de varias formas al modelo.

Esta situación esta resumida en el siguiente cuadro.

	Dinámica.		Estática	
	Lineal.	No lineal.	Lineal	No lineal.
Determinística	Modelo dinámico de Leontieff. Sistemas dinámicos.		Modelo estático de Leontieff. Programación lineal. Redes.	Programación convexa. Programación no convexa.
Probabilística		Modelo general de programación dinámica. Teoría de inventarios	Programación lineal estocástica. Juegos de suma cero.	

Figura 1.1. Programación Matemática.

Hay muchas otras formas de abordar problemas de optimización. Algunas de ellas se mencionan a continuación.

Método analítico.

Es el método base del Cálculo Diferencial, en el se considera que sí existe la función F de una sola variable y es derivable en todo su rango, se pueden hallar todos sus mínimos y máximos, sean locales o globales. Formalmente tenemos los siguientes teoremas:

Si una función f es continua en un intervalo cerrado [a, b], entonces f alcanza un mínimo y un máximo por lo menos una vez en [a, b].
Si una función f es continua en un intervalo cerrado [a, b] y alcanza su máximo o su mínimo en un número c del intervalo abierto (a, b), entonces f ’(c) = 0 o f ’(c) no existe.
Un número c en el dominio de una función f se llama número crítico de f si f ’(c) = 0 f ‘(c) no existe.

Los pasos para determinar los máximos o mínimos de una función son:

1. Encontrar todos los números críticos de f.

2. Calcular f (c) para cada número crítico c.
3. Calcular f(a) y f(b).
4. El máximo y el mínimo absolutos de f en [a, b] son, respectivamente, el mayor y el menor de los valores de la función determinados en los pasos 2 y 3.

Sin embargo, se presentan problemas al considerar que a veces puede ser muy difícil determinar los números críticos de una función, de hecho estos podrían no existir, la función f en muchos casos puede ser desconocida, y en caso de que se conozca no tiene porqué ser diferenciable, además el tratamiento analítico para funciones de más de una variable suele ser muy complicado.

Métodos exhaustivos, aleatorios y heurísticos.

Los métodos exhaustivos son aquellos que recorren todo el espacio de búsqueda, quedándose con la mejor solución, por supuesto que su principal desventaja es el hecho de tener que realizar un recorrido completo.

Fueron propuestos para un modelo lineal por Jutler y Solich y utilizados por Osyczka, Rao y Tseng Lu. Se suelen usar en juegos para examinar las posibilidades a partir de la jugada actual. Algoritmos como el MINIMAX ejemplifican este tipo de algoritmos, en el método MINIMAX se realiza una exploración exhaustiva del árbol de búsqueda, de manera que el jugador que debe mover en primer lugar (MAX) tiende a elegir en cada uno de sus movimientos aquel camino que le conduzca a un nodo frontera con el mayor valor posible de la función de evaluación (si se aplica el método MAX). La estrategia del otro jugador (MIN) es la contraria, ya que trata de contrarrestar las decisiones de MAX, eligiendo el peor valor para aquella, es decir el más pequeño.

Los métodos heurísticos utilizan reglas para eliminar zonas del espacio de búsqueda consideradas “poco interesantes”. Se consideran estrategias de resolución y de decisión utilizadas por los especialistas, basados en la experiencia previa con problemas similares. Estas estrategias indican las vías o posibles enfoques a seguir para alcanzar una solución.

De acuerdo con Monero y otros (1995) los procedimientos heurísticos son acciones que comportan un cierto grado de variabilidad y su ejecución no garantiza la consecuencia de un resultado óptimo como, por ejemplo reducir el espacio de un problema complejo a la identificación de sus principales elementos.

Mientras que Duhalde y González (1997) señalan que un heurístico es “*un procedimiento que ofrece la posibilidad de seleccionar estrategias que nos acercan a una solución*”. En todo caso los métodos heurísticos son aquellos que proporcionan una buena solución, no necesariamente la óptima, en un tiempo razonable; los métodos heurísticos dependen mucho del problema en particular para el cual se diseñan.

En los métodos aleatorios, se va muestreando el espacio de búsqueda acotando las zonas que no han sido exploradas; se escoge la mejor solución, y además se da el

intervalo de confianza de la solución encontrada. El método Montecarlo es un ejemplo de este tipo de métodos.(2)

El método Montecarlo consiste en tomar una solución válida, provocar una pequeña variación aleatoria y si la nueva configuración obtiene mejores resultados que la anterior y cumple las restricciones, conservar la nueva; de otra manera mantener la anterior. Este planteamiento no permite que el método converja, ya que aunque llegue al mínimo sigue iterando. Así que para buscar que converja se va haciendo cada vez más pequeña la variación aleatoria. Si bien el método de Montecarlo puede no converger en un mínimo el coste es razonable y puede ser aceptado como solución para problemas de mucha complejidad. Otra optimización tradicional consiste en provocar la variación aleatoria de forma que la nueva configuración a desarrollar ya sea válida, esto es que cumpla con las restricciones, mas esto no es siempre posible.

Método escalador.

También llamado en inglés hillclimbing (subiendo la montaña o el cerro), en estos métodos se va evaluando la función en uno o varios puntos, pasando de un punto a otro en el cual el valor de la evaluación es superior. La búsqueda termina cuando se ha encontrado el punto con un valor máximo. El algoritmo se resume en los siguientes pasos:

Algoritmo escalador.

1. Escoger una solución inicial (X_j, ..., X_n)
2. Mientras que siga subiendo el valor de F, hacer
   3. Alterar la solución (X'_j, ..., X'_n) = (X_j, ..., X_n) + (Y_j, ..., Y_n) y evaluar F
   4 Si F(X'_j, ..., X'_n) > F(X_j, ..., X_n), hacer (X_j, ..., X_n) = (X'_j, ..., X'_n)
   5. Volver a 2.

Estos algoritmos toman muchas formas diferentes, según el número de dimensiones del problema, el valor del incremento y la dirección que se tiene que dar. En algunos casos se utiliza el método Montecarlo para elegir la solución de forma aleatoria.

El principal problema de este tipo de algoritmos es que tienden a quedarse en el máximo local más cercano a la solución inicial, y no son validos para problemas multimodales, en los cuales la función de costo tiene varios posibles óptimos.

Recocimiento simulado.

Conocido en inglés como Simulated Annealing, el nombre viene de la forma de conseguir ciertas aleaciones; una vez fundido el metal se va enfriando poco a poco para conseguir la estructura cristalina correcta. Se le podría calificar como escalador estocástico, y su principal objetivo es evitar los mínimos locales en los que caen los

(2) Herran Manu Algoritmos Genéticos www.aircenter.net/gaia

escaladores, para ello, no siempre toma la solución óptima, sino que en algunos casos puede escoger una solución cercana a la óptima, siempre que la diferencia entre ambas tenga un nivel determinado, que depende de un parámetro llamado "temperatura".

Sus pasos son los siguientes:

Algoritmo de recocimiento simulado.

1. Iniciar la temperatura T, la solución inicial (X_j, ..., X_n) y evaluar F(X_j, ..., X_n).
2. Repetir los pasos siguientes hasta que la temperatura sea nula o el valor de F converja:
   3. Disminuir la temperatura.
   4. Seleccionar una nueva solución (X_j, ..., X_n) = (X'_j, ..., X'_n) en la vecindad de la anterior y evaluarla.
   5. Si F(X'_j, ..., X'_n) > F(X_j, ..., X_n), hacer (X_j, ..., X_n) = (X'_j, ..., X'_n), si no, generar un número aleatorio R entre 0 y 1.
      Si $\exp\left(\frac{F(Xj, ..., Xn) - F(Xj, ..., Xn)}{T}\right) > R$, entonces ($X_j$, ..., X_n) = (X'_j, ..., X'_n)

Técnicas basadas en población.

La mayoría de los problemas son comúnmente abordados por medio de algoritmos clásicos, algoritmos recursivos o de tipo voraz, sin embargo existen otro tipo de problemas, los llamados NP-completos en los que la complejidad crece con el tamaño del problema de forma exponencial, en ellos no es posible usar estos algoritmos. El problema del agente viajero, en el cual dada una serie de ciudades y la distancia que las separa hay que calcular un camino tal que la distancia total de recorrido sea mínima, y no se tenga que visitar ninguna ciudad más de una vez, es paradigmático de este tipo de problemas. Para este tipo de problemas se han ensayado diversos métodos, entre ellos las técnicas basadas en poblaciones.

Este tipo de técnicas pueden ser una versión de las anteriores, pero en vez de tener una sola solución, que se va alterando para obtener el óptimo, se persigue el óptimo cambiando varias soluciones, de esta forma es más fácil escapar de los mínimos locales. Entre estas técnicas se encuentran la mayoría de los algoritmos de la informática evolutiva.

1.2 Teoría de la evolución.

Los algoritmos genéticos, se encuentran dentro de las técnicas basadas en poblaciones, pero ¿Cómo surgió la idea de imitar a la naturaleza para hacer evolucionar una solución? La base teórica de los algoritmos genéticos se encuentra en la Teoría de la Evolución de las Especies. El objetivo de esta sección es presentar un rápido repaso del desarrollo de dicha teoría, mediante el estudio de sus antecedentes históricos,

precursores y creadores de la misma; así como la explicación de los puntos básicos de la evolución.

La naturaleza y el fenómeno natural llamado evolución son la inspiración de los algoritmos genéticos. La teoría de la evolución tiene sus antecedentes en el filósofo griego Anaximandro (611- 547 AC.) y el romano Lucrecio (99- 55 AC) quienes acuñaron el concepto de que todos los seres vivos se encuentran relacionadas y cambian en el transcurso del tiempo. La ciencia en su época se basaba principalmente en la observación por lo que resulta sorprendente lo acertado de sus observaciones con respecto a la teoría actual.

Otro filósofo griego, Aristóteles desarrollo su "*Scala Naturae*" o Escala de la naturaleza, para explicar su concepto del avance de los seres vivientes, desde lo que él consideraba inanimado como las plantas, luego los animales y finalmente el hombre. Este concepto del hombre como "*cumbre de la creación*" aun subsiste hoy, sin ningún fundamento lógico.

Durante la edad media, el pensamiento de los científicos post-Aristotélicos fue restringido por la ideología judeocristiana que exigía la aceptación de las escrituras del libro del Génesis, perteneciente al viejo testamento, en él se expone el concepto de creación del mundo en seis días por la acción divina. A mediados del siglo XVII el arzobispo irlandés James Ussher, calculó la edad de la tierra basado en la genealogía desde Adán y Eva de acuerdo al Génesis yendo hacia atrás desde la crucifixión de Jesús de Nazaret. Sus cálculos concluyeron que la tierra se formó el 22 de octubre del 4004 AC. Estos cálculos formaron parte de su libro sobre la historia del mundo y la cronología que desarrolló fue impresa en la primera página de la Biblia. Las ideas de Ussher fueron aceptadas rápidamente y sin discusión.

Los pioneros de la Geología dudaron de la verdad de una tierra de 5000 años de antigüedad. Leonardo da Vinci calculó, en base a los sedimentos del río italiano Po, que debió tomar al menos 200 000 años para que se formaran dichos depósitos. Esto ocurrió en una época en que científicos de la talla de Galileo Galilei eran convictos por la inquisición bajo el cargo de herejía por haber sostenido que la tierra no era el centro del universo, por lo que el avance científico se vio obstaculizado por siglos.

No sería sino hasta 1795 que James Hutton, considerado el padre de la Geología moderna desarrolló la teoría del uniformismo, base tanto de la geología como de la paleontología moderna. De acuerdo con Hutton, ciertos procesos geológicos operaron en el pasado de la tierra del mismo modo que actúan hoy en día. Por lo tanto muchas estructuras geológicas no pueden ser explicadas con una tierra de solo 5000 años. El geólogo británico Sir Charles Lyell refinó las ideas de Hutton durante el siglo XIX, y concluyó que el efecto lento, constante y acumulativo de las fuerzas naturales había producido un cambio continuo en la tierra, su libro "Los principios de la Geología" tuvo un efecto profundo en Charles Darwin y Alfred Wallace.

Erasmus Darwin (1731-1802) abuelo de Charles Darwin; médico y naturista británico, propuso que la vida había cambiado, pero no presentó un mecanismo claro de cómo

ocurrieron dichos cambios, sin embargo sus notas son interesantes por la posible influencia que pueden haber tenido sobre su nieto. Mientras que George-Louis Leclerc (1707-1788); propuso que las especies (pero solo las que no habían sido el producto de la creación divina) pueden cambiar. Su trabajo supuso un gran avance sobre el primitivo concepto de que todas las especies se originan por un creador perfecto y por lo tanto no pueden cambiar debido a su origen divino, etc.

El botánico sueco Carl Von Linneo o Linné (1707-1778), intentó clasificar todas las especies conocidas en su tiempo en categorías inmutables. Muchas de esas categorías todavía se usan en Biología actual. La clasificación jerárquica de Linneo se basaba en la premisa de que las especies eran la menor unidad clasificable, y que cada especie (o taxón) estaba comprendida dentro de una categoría superior.

William Smith (1769-1839), empleado de la industria minera inglesa, desarrolló el primer mapa geológico preciso de Inglaterra. También concibió durante sus viajes el principio de la sucesión biológica. La idea sostiene que cada periodo de la historia de la tierra tiene su particular registro fósil. En esencia Smith dio inicio a la ciencia de la Estratigrafía, la identificación de las capas de roca basada, entre otras cosas por su contenido fósil.

Georges Cuvier (1769-1832), brillante paleontólogo propuso la teoría de las catástrofes para explicar la extinción de las especies. Pensaba que los eventos geológicos dieron como resultado grandes catástrofes. Y que la existencia de varias creaciones ocurrió después de cada catástrofe. Louis Agassiz (1807-1873) propuso entre 50 y 80 catástrofes seguidas de creaciones nuevas e independientes. Aunque estas teorías actualmente no son creíbles eran bastante confortables para la época y fueron ampliamente aceptadas.

Jean Baptiste de Lamarck (1744-1829) el científico que acuñó el término Biología, concluyó audazmente, que los organismos mas complejos evolucionaron de organismos mas simples preexistentes. Propuso la herencia de los caracteres adquiridos para explicar, entre otras cosas, el largo del cuello de la jirafa. El trabajo de Lamarck dio vida a una teoría que señalaba la existencia de cambios en las especies en el tiempo debido al uso o desuso de sus órganos y postuló un mecanismo para ese cambio.

Charles Darwin y Alfred Wallace, ambos trabajando por separado, realizaron extensos viajes y, eventualmente, desarrollaron la misma teoría acerca de cómo cambió la vida a lo largo de los tiempos; así como el mecanismo para ese cambio: **La selección natural**.

Charles Darwin (1809-1882) obtuvo a sus veintidós años una plaza en "*El Beagle*", barco al mando de Robert Fitzroy, este viaje dio a Darwin la oportunidad única de estudiar la adaptación y obtener un sinnúmero de evidencias que fueron utilizadas en su teoría de la evolución. Darwin dedicó mucho tiempo a coleccionar especimenes de plantas, animales y fósiles y a realizar extensas observaciones geológicas. El viaje incluyó entre otros puntos, toda la costa atlántica sudamericana y el paso por el estrecho de Magallanes. Una de las escalas mas importantes durante su viaje ocurrió en el archipiélago de la Galápagos, frente a Ecuador, en cuyas áridas islas observó a las

especies de pájaros pinzones, las famosas tortugas gigantes y notó sus adaptaciones a los diferentes hábitat isleños. Al regresar a Inglaterra en 1836, comenzó a catalogar su colección y a fijar varios puntos de su teoría; siendo los más importantes los siguientes:

Adaptación: Todos los organismos se adaptan a su medio ambiente.

Variación: Todos los organismos presentan caracteres variables, ellos son una cuestión de azar, aparecen en cada población natural y se heredan entre los individuos. No las produce una fuerza creadora, ni el ambiente, ni el esfuerzo inconsciente del organismo, no tienen destino ni dirección, pero a menudo ofrecen valores adaptativos positivos o negativos.

Sobre-reproducción: Todos los organismos tienden a reproducirse mas allá de la capacidad de su medio ambiente para proporcionales sustento (esta idea se basó en las teorías de Thomas Malthus, quien señaló que las poblaciones tienden a crecer geométricamente hasta encontrar un limite al tamaño de su población, que viene dado por las restricciones de su ambiente, entre otras la cantidad de alimento, superficie, etc). Dado que no todos los individuos están adaptados por igual a su medio ambiente, algunos sobrevivirán y se reproducirán mejor que otros, esto es conocido como "*selección natural*". Algunas veces se hace referencia a este hecho como "*la supervivencia del más fuerte*", en realidad tiene más que ver con los logros reproductivos del organismo que con la fuerza del mismo.

Figura 1.2 Charles Darwin

El inglés Alfred Russel Wallace (1823-1913) paso muchos años en Sudamérica, publicó sus notas en el libro "*Viajes en el Amazonas y el Río Negro*" en 1853, al año siguiente dejaría Inglaterra para estudiar la historia natural de Indonesia y escribiría sus ideas acerca de la selección natural. Podemos percibirlas en el siguiente fragmento: "*... que la perpetua variabilidad de todos los seres vivos tendría que suministrar el material a partir del cual, por la simple supresión de aquellos menos adaptados a las condiciones del medio, sólo los más aptos continuarán en carrera...*".

El trabajo de Wallace, publicado en 1858, fue el primero en definir el rol de la selección natural en la formación de las especies. En conocimiento del mismo, Darwin se apresuró a publicar, en noviembre de 1859 su mayor tratado, "*El origen de las especies*". En base a lo relatado, si bien la teoría de la evolución se atribuye generalmente a Darwin, para ser correcto es necesario mencionar que ambos, Darwin y

Wallace, desarrollaron la teoría y llegaron a conclusiones similares por separado. Dicha teoría puede ser resumida en el hecho de que pequeños cambios heredables en los seres vivos y la selección son los dos hechos que provocan el cambio en la naturaleza y la generación de nuevas especies.

Ni Darwin, ni Wallace consiguieron explicar de forma convincente cómo ocurre la evolución, es decir cómo pasaban las variaciones de una generación a otra; correspondería a otros dicha tarea.

1.3 Selección natural y genética.

La teoría de la evolución explica cómo cambian las especies a través de las épocas, sin embargo faltaba la explicación del mecanismo que permite la evolución, es decir cómo se transmitían las características de una generación a otra; en esta sección se muestra la manera en que la genética se desarrolló como ciencia y complementó la teoría de la evolución, convirtiéndola en un hecho comprobable, que más tarde inspiraría a los creadores de algoritmos a utilizarla como base para resolver problemas.

Fue Gregor Mendel (1822-1884) quien descubrió que los caracteres se heredaban de forma discreta, y que se tomaban del padre o de la madre, dependiendo del carácter dominante o recesivo. A estos caracteres que podían tomar diferentes valores se les llamó genes, y a los valores que podían tomar se les llamo alelos. Tristemente las teorías de Mendel (quien trabajó en total aislamiento) no recibirían atención, Darwin no llegó a conocerlo, y tras de realizar sus investigaciones fue nombrado abad de su monasterio y a ello dedicó el resto de su vida. Redescubiertos a principios del siglo XX, los trabajos de Mendel proveerían las respuestas necesarias. La combinación de la genética mendeliana y la teoría de la evolución de Darwin se conoce como teoría NEODARWINIANA o Teoría Sintética de la Evolución.

Figura 1.3 Gregor Mendel

Sería hasta 1930 que el genetista inglés Robert Aylmer relacionó ambas teorías, demostrando que los genes mendelianos eran los que proporcionaban el mecanismo necesario para la evolución. Paralelamente el biólogo alemán Walter Flemming describió los cromosomas, como ciertos filamentos en los que se agregaba la cromatina del núcleo celular durante la división; poco más adelante se descubrió que las células de cada especie viviente tenían un número fijo y característico de cromosomas.

Sería en los años cincuentas, cuando Watson y Crick descubrieron que la base molecular de los genes está en el ADN (ácido desoxirribonucleico), los cromosomas están compuestos de ADN, y por lo tanto los genes están en los cromosomas. La macromolécula de ADN está compuesta por bases púricas y pirimidínicas, la adenina, citosina, guanina y tiamina. La combinación y secuencia de estas bases forma el código genético, único para cada ser vivo. Grupos de tres bases forman un codon, y cada codon codifica una proteína; el código genético codifica todas las proteínas que forman parte de un ser vivo. Mientras que al código genético se le llama genotipo, al cuerpo que construyen esas proteínas, modificado por la presión ambiental y la historia vital, se le llama fenotipo.

Figura 1.4 Cromosomas

No toda la cadena de ADN codifica proteínas; las zonas que codifican proteínas se llaman intrones, las zonas que no lo hacen exones. Un gen comienza con el sitio tres o aceptor y termina con el sitio cinco o donante. Proyectos como el del genoma humano tratan de identificar cuáles son estos genes, sus posiciones y sus posibles alteraciones, que habitualmente conducen a enfermedades.

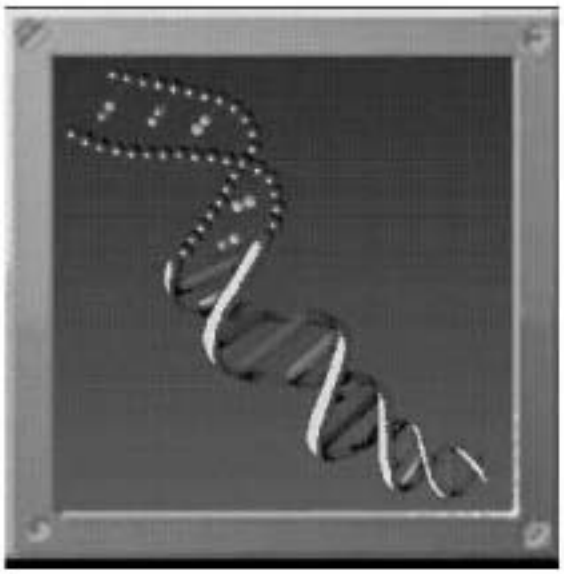


Figura 1.5 Cadena de ADN

La teoría del Neo-darwinismo afirma que la historia de la vida fue causada por una serie de procesos que actúan en y dentro de las poblaciones; la reproducción, mutación, competición y selección son algunos de estos procesos. La evolución se puede definir entonces como cambios en el "pool" o conjunto genético de una población.

La evolución cuenta con una serie de mecanismos de cambio, los que será necesario entender para comprender los algoritmos genéticos, ya que el concepto de estos consiste en imitar dichos mecanismos para resolver problemas de ingeniería. Los mecanismos de cambio alteran la proporción de alelos de un tipo determinado en una población, y se dividen en dos tipos: los que disminuyen la variabilidad, y los que la aumentan.

Los principales mecanismos que disminuyen la variabilidad son:

Selección natural: Los individuos que tengan algún rasgo que los haga menos válidos para realizar sus funciones como seres vivos, no llegarán a reproducirse, y por tanto su patrimonio genético desaparecerá del pool, algunos no llegarán siquiera a nacer. Esta selección sucede a diversos niveles: competición entre miembros de la especie, competición entre diferentes especies y competición predador-presa. También es importante la selección sexual, en la cual las hembras eligen el mejor individuo para reproducirse.

Deriva genética: El simple hecho de que un alelo sea más común en la población que otro, causará que la proporción de alelos de esa población vaya aumentando en una población aislada, lo cual a veces da lugar a fenómenos de especialización.

Otros mecanismos que aumentan la diversidad y suceden a nivel molecular son:

Mutación: La mutación es una alteración del código genético, que puede suceder por múltiples razones. En muchos casos las mutaciones que cambian un nucleótido por otro

son letales, y los individuos ni siquiera llegan a desarrollarse, pero a veces se da lugar a la producción de una proteína que aumenta la supervivencia del individuo, y que por lo tanto es heredada a la descendencia. Las mutaciones son totalmente aleatorias, y son el mecanismo básico de generación de variedad genética.

Poliploidía: Mientras que las células normales poseen dos copias de cada cromosoma, y las células reproductivas uno (haploides), pueden suceder por accidente que alguna célula reproductiva tenga dos copias; si se logra combinar con otra célula diploide o haploide dará lugar a un ser vivo con varias copias de cada cromosoma. La poliploidía da lugar a individuos con algún defecto, pero en algunos casos se llega a generar individuos viables.

Recombinación: Cuando dos células sexuales, o gametos, una masculina y otra femenina se combinan, los cromosomas de cada una también lo hacen, intercambiándose genes, que a partir de ese momento pertenecerán a un cromosoma diferente. En ocasiones también se produce traslocación dentro de un cromosoma; es decir una secuencia de código se elimina de un sitio y aparece en otra posición del cromosoma, o en otro cromosoma.

Flujo genético: Es el intercambio de material genético entre seres vivos de diferentes especies. Normalmente se produce a través de un vector, que suelen ser virus o bacterias; estas incorporan a su material genético genes procedentes de una especie a la que han infectado, y cuando infectan a un individuo de otra especie pueden transmitirle esos genes a los tejidos generativos de gametos.

En resumen la selección natural actúa sobre el fenotipo y suelen disminuir la diversidad, haciendo que sobrevivan sólo los individuos más aptos, los mecanismos que generan diversidad y que combinan características, actúan sobre el genotipo.

1.4 Informática evolutiva.

Después de describir los mecanismos de la evolución, se presenta ahora la historia del cómo evoluciono la idea de simular o imitar a la evolución con el objeto de resolver problemas humanos; esto se consigue mediante el repaso histórico de los precursores de esta idea y de los investigadores que le dieron el enfoque actual a la teoría.

Las primeras ideas, son anteriores incluso al descubrimiento del ADN y vinieron de Von Neumann, quien afirmaba que la vida debía estar apoyada por un código que a la vez describiera cómo se puede *construir* un ser vivo de tal manera que dicho ser fuera capaz de reproducirse. Por lo tanto un autómata o máquina auto reproductiva tendría que ser capaz de reproducirse y además de contener las instrucciones que le permiten hacerlo, y adicionalmente poder copiar tales instrucciones a su descendencia.

Fue a mediados de 1950, cuando el rompecabezas de la evolución casi se había completado que Box, trató de imitarla mediante su técnica EVOP (Evolutionary Operation), que consistía en elegir una serie de variables, que rigen un proceso industrial, crear pequeños cambios para formar un hipercubo, cambiando los valores de

las variables en una cantidad fija. Se probaba entonces con cada una de las esquinas del hipercubo durante un tiempo, y al final del periodo de pruebas, un comité humano decidía sobre la calidad de los resultados. Es decir que aplicaba mutación y selección a las variables para mejorar la calidad del proceso.

En 1958 Friedberg, trató de mejorar las técnicas evolutivas mediante un programa, diseñado en código máquina de 14 bits; cada programa tenía 64 instrucciones. Un programa llamado **Herman**, ejecutaba los programas creados, y otro programa llamado **Teacher** o profesor, le ordenaba a **Herman** ejecutar otros programas y ver si los ya ejecutados habían cumplido su labor. Los programas recibían su labor en una posición de memoria y depositaban el resultado en otra, que era examinada al terminar la ejecución. Para hacer evolucionar los programas Friedberg hizo que en cada posición de memoria hubiera dos alternativas; para cambiar un programa, alternaba las dos instrucciones, o bien las reemplazaba con una nueva totalmente aleatoria. Se puede concluir que usaba mutación para generar nuevos programas, sin embargo tuvo poco éxito, ya que la mutación sin selección ocasiona que la búsqueda sea prácticamente aleatoria.

Bremmerman trató de usar la evolución para "*entender los procesos del pensamiento creativo y aprendizaje*", [(3)] y empezó a considerar la evolución como un proceso de aprendizaje. Para resolver un problema, codificaba las variables del problema en cadenas binarias de ceros y unos; luego sometía la cadena a mutación cambiando un bit cada vez, de manera que concluyó que la tasa ideal de mutación era de un bit. Bremmerman trató de resolver problemas de minimización de funciones, aunque no tuvo claro el proceso de selección, el tamaño y tipo de su población. Sus estudios llegaron a un punto llamado la "*trampa Bremmerman*", en el cual la solución ya no mejora, trato de añadir cruce de soluciones, sin éxito. Una vez más, el uso de operadores que generan diversidad mostró no ser suficiente para dirigir la búsqueda genética a la solución correcta.

El primer uso de procedimientos evolutivos en inteligencia artificial se debe a Reed, Toombs y Baricelly, que trataron de hacer evolucionar un autómata que jugaba a un caso simplificado de cartas. Las tácticas del juego se basaban en una serie de estrategias, de acuerdo con la mano obtenida, con cuatro parámetros de mutación asociados. Se mantenía una población de 50 individuos y aparte de la mutación, había intercambio de probabilidades entre dos padres, se eliminaban a los perdedores, sin embargo los resultados no fueron satisfactorios y concluyeron que el entrecruzamiento no aportaba mucho a la búsqueda.

Estos experimentos demostraron el potencial de la evolución como método de búsqueda de soluciones novedosas.

Aunque el proceso para solucionar un problema es similar, en los algoritmos evolutivos se distinguen tres maneras de hacerlo. La diferencia principal se da en el nivel en el cual

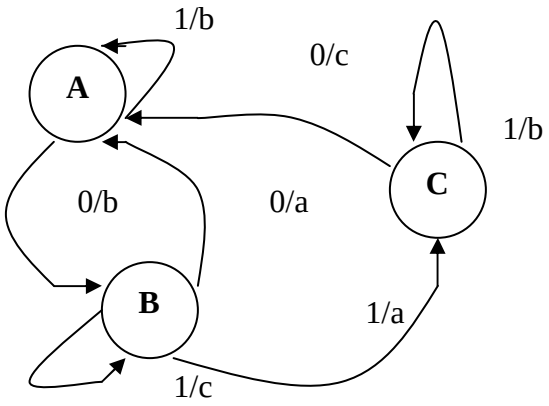
[(3)] Guervós Merelo J Informática Evolutiva. http://kal-el.urg.es

se simula la evolución, en sus operadores primarios y secundarios y en el nivel de representación con el cual trabajan.

1) Programación evolutiva:

Fue originalmente propuesta por Fogel en 1960, en ella remarcaba los nexos hereditarios y de comportamiento entre padres y sus descendientes, la adaptación se concibe en esta técnica como una forma de inteligencia. Se inició como un intento de usar la evolución para crear máquinas inteligentes, que pudieran detectar su entorno y reaccionar adecuadamente a él. Para esto utilizó un autómata celular. Un autómata celular es un conjunto de estados y las reglas de transición entre ellos, de forma que al recibir una entrada, se cambia o no de estado y se produce una salida.

Fogel trataba de hacer aprender a los autómatas a encontrar regularidades en los símbolos que se le iban enviando. Como método de aprendizaje usó un algoritmo evolutivo, ya que la población de autómatas competía para hallar la mejor solución, es decir predecir cual sería el siguiente símbolo en la secuencia con un mínimo de error, el 50% de los peores autómatas eran eliminados cada generación, y sustituidos por autómatas resultantes de una mutación de los restantes. El modelo de la programación Evolutiva ocurre a nivel de especie, por lo tanto no existe un operador genético de cruce. La selección de individuos se realiza de manera probabilística y el operador genético es la mutación, además la programación evolutiva opera a nivel fenotipo.



Ejemplo de autómata celular como los usados por Fogel, con tres estados.
Los caracteres encima de las líneas que unen los estados indican la entrada y la salida correspondiente.
Figura 1.6. Autómata celular

El algoritmo básico de la programación evolutiva es el siguiente:

1. Generar aleatoriamente una población inicial.
2. Aplicar mutación.
3. Calcular aptitud de todos los individuos de la población.
4. Seleccionar mediante torneo (normalmente probabilístico) los individuos que sobrevivirán.

Las aplicaciones más comunes de la programación evolutiva son las siguientes: Predicción, Generalización, Juegos, Control Automático, Planeación de Rutas, Diseño y entrenamiento de Redes Neuronales y Reconocimiento de Patrones.

2) Estrategias Evolutivas:

A mediados de los años 60, Rechenberg y Schwefel [(4)]desarrollaron en Alemania las Estrategias de Evolución, con el objeto de resolver problemas de Aerodinámica con alto grado de complejidad. Consisten en métodos de optimización paramétricos que trabajan sobre poblaciones de cromosomas representados con números reales. La más común de estas estrategias crea nuevos individuos a partir de la población actual, añadiendo un vector de mutación a los cromosomas existentes en la población; en cada generación se elimina un porcentaje de la población y con los individuos restantes se crea la nueva población total mediante mutación y cruce. La magnitud del vector mutación se calcula adaptativamente.

Las Estrategias de Evolución trabajan con una abstracción a nivel de individuo, por lo que el operador de cruce es del tipo sexual (dos padres) o panmítica (un solo padre), sin embargo si llega a ser usado se considera como un operador secundario. La mutación es el operador primario y se utiliza con valores Gaussianos. Estos valores son los que permiten que las estrategias de evolución sean auto-adaptativas, ya que el valor de mutación varía en el tiempo. El nivel de operación de las estrategias de evolución es fenotípico. Su selección es determinística y extintiva (los peores individuos tienen probabilidad cero de sobrevivir).

También se pueden considerar las estrategias de evolución como algoritmos de optimización numérica, que tratan de hallar una serie de parámetros, $(X_1, X_2, ..., Xn)$, tales que $f(X_1, X_2, ..., Xn)$ sea mínimo. Las estrategias de evolución mantienen una población de vectores, que evolucionan mediante los operadores:

Mutación: A cada $(X_1, X_2, ..., Xn)$ se añade un vector aleatorio de distribución normal con una varianza o, que varía durante el algoritmo.

Recombinación: Se intercambia parte del vector.

La versión original, llamada (1 + 1) – EE (Estrategias de Evolución) no contemplaba el concepto de población. Existe un solo padre y a partir de él se genera un nuevo individuo mediante la siguiente expresión:

$$X^{t+1} = X^t + N(0, \sigma)$$

donde t se refiere a la generación y $N(0, \sigma)$ es un vector de números Gaussianos independientes con media cero y desviación estándar σ. Si el hijo es mejor en aptitud se mantiene de lo contrario se elimina.

[(4)] Guervós Merelo J Informática Evolutiva. http://kal-el.urg.es

La estrategia de evolución (μ + 1) ya utiliza una población. Hay μ padres y se genera un solo hijo, el cual puede reemplazar al peor padre. Schwefel introduciría en 1975 el uso de múltiples hijos en las estrategias de evolución llamadas (μ + λ) y (μ, λ). En la primera, los μ mejores individuos obtenidos de la unión de padres e hijos se preservarán (población traslapable). En la segunda, sólo los mejores μ hijos de la siguiente generación sobreviven (población no traslapable).

Así las estrategias de evolución se clasifican dependiendo del método de selección y reciben diferentes denominaciones con las letras μ y λ de la siguiente forma:

- (1 + 1) Sólo hay un padre y un hijo; el hijo sustituye al padre si su fitness es mejor.
- (μ + 1) Se mantienen en la población μ padres, y se genera un hijo cada vez. Si tiene un fitness más alto, sustituye a uno de los padres.
- (μ, λ) λ hijos sustituyen a μ padres, caiga quien caiga. Al parecer, esta es la estrategia preferida por Schwefel.
- (μ + λ) En esta, los padres y los hijos conviven; se crean μ padres, a partir de ellos se crean λ hijos; se evalúan juntos, y quedan los μ mejores para la siguiente generación.

Las Estrategias de Evolutivas se han aplicado a problemas de ruteo y redes, Bioquímica, Óptica, diseño en Ingeniería y magnetismo.

3) Programación Genética:

Consiste en hacer evolucionar programas de computadora siguiendo los principios de la evolución. Dado que por reproducción, mutación y cruce se puede optimizar casi todo; John Koza pensó en aplicar estos operadores a programas de computadora.[5] Sin embargo hay una serie de problemas a considerar: los programas de computadora tiene una sintaxis muy rígida y si se aplica una mutación a algún programa lo más probable es que resulte un programa no válido. Si se hacen los cambios en nivel de código máquina, la probabilidad de mutación letal es casi del 100%, dado que si se modifican las posiciones de memoria a las que se accede, el programa entrará probablemente en un bucle infinito o quedara trabado irremisiblemente.

Según Koza, el representar las soluciones de un programa mediante otro programa tiene una serie de ventajas:

- La solución al problema lleva ya implícito el algoritmo que lo soluciona, no es necesario fijar el algoritmo y optimizar los parámetros del mismo.
- El representar una solución mediante un programa es más natural que usar una serie de números reales o una cadena binaria.

[5] Koza John Genetic programming: On the programming of Computers by means of Natural Selection.

- Las soluciones no están predeterminadas ni en forma ni en tamaño. No todos los algoritmos genéticos usan cromosomas de longitud fija, pero la interpretación de esos cromosomas si está fija de antemano.

Para evitar los problemas de la representación clásica y hacer posible la evolución de los programas tal como Friedberg había intentado realizar con anterioridad, se hacia necesario crear, diseñar o adaptar un lenguaje de programación de tal forma que todos los cambios posibles que pudieran ocurrir por causa de la evolución fueran válidos, es decir que no se presenten errores. La vía principal para lograr esto fue tomar un número limitado de operadores y de variables a las que se puede acceder, de forma que cualquier mutación, al actuar sobre estas variables, genere otras variables u otros operadores. Esta es la opción usada en el sistema Tierra de Tom Ray, el sistema trabaja en código máquina, pero si se desea utilizar un lenguaje de alto nivel, las mutaciones, además de conservar el léxico, tiene que conservar la sintaxis, es decir la forma en que se ordenan los símbolos entre sí; esto hace que se requiera tratar las mutaciones de forma especial, o simplemente eliminarlas. Lo más probable es que el cruce rompa la sintaxis de una frase, lo que hace necesario utilizar un tipo especial de cruce que conserve la estructura.

La programación genética encontraría en el lenguaje de programación LISP las siguientes ventajas para este trabajo:

- Hay una correspondencia directa entre la sintaxis y la estructura en árbol que resulta del análisis sintáctico de la expresión, lo cual hace fácil pasar de una a la otra.
- LISP trata programas y datos de la misma forma; un programa puede crear y manipular otro programa fácilmente, y ejecutarlo mediante la orden *eval*.
- LISP permite manipular estructuras dinámicas, y alterar la forma del programa en tiempo de ejecución.

Los pasos a ejecutar para hacer evolucionar programas mediante la programación genética son:

1. Generar una población inicial de programas mediante composición aleatoria de las funciones y terminales del problema; las cuales previamente se han seleccionado.
2. Ejecutar cada programa de la población y asignarle un valor de fitness dependiendo de la eficiencia con que resuelva el problema.
3. Crear una nueva población aplicando las siguientes operaciones primarias.
   - Copiar programas existentes a la nueva población.
   - Crear nuevos programas recombinando genéticamente partes elegidas aleatoriamente de dos programas ya existentes.
4. Repetir los pasos dos y tres hasta satisfacer el criterio de terminación.
5. Elegir como resultado el mejor programa que haya aparecido en cualquiera de las generaciones.

Hay varias formas de crear las estructuras iniciales; dado que la mutación prácticamente no se usa, toda la diversidad necesaria para resolver el problema debe de estar en la población inicial, las principales maneras de generar esta diversidad son:

- Full: Se crean programas cuyos árboles tienen todos la máxima profundidad permitida, y están completos, es decir todos tiene dos símbolos terminales.
- Grow: Se generan árboles de programa, con cualquier tamaño menor o igual que el máximo, los árboles no tienen porque estar equilibrados.
- Ramped half & half: Crea árboles completos, pero para cada una de las profundidades posibles. Al parecer esta opción es la que da mejores resultados.

La selección que usa la programación genética es similar a la que se observara en los algoritmos genéticos, excepto que se suele preferir una selección basada en el orden, y la reproducción por estado estacionario.

Otros operadores que se usan ocasionalmente en programación genética son:

Mutación: Consiste en cambiar un nodo del árbol y todo lo que desciende de él por otro subárbol generado aleatoriamente.

Permutación: Consiste en intercambiar la posición de los subárboles de un nodo.

Editado: Se usa para simplificar los programas resultantes; consiste en simplificar analíticamente las expresiones que aparecen en el programa lo que permite que las soluciones sean más compactas, aunque no está muy clara la ventaja de realizarlo durante la ejecución del algoritmo frente a la opción de simplificar cuando termina la ejecución de esté.

Encapsulado: Consiste en sustituir un subárbol por un solo símbolo, de forma que el subárbol aparezca siempre junto en las operaciones de recombinación. Viene a ser un seguro ante la ruptura de un subárbol beneficioso.

Aplicaciones de la computación Evolutiva

Las aplicaciones de la programación genética son muy amplias, en realidad todo lo que se pueda describir con un programa en LISP se puede programar genéticamente. Sin embargo, no está muy claro cuál es la ventaja frente a la simple programación de un algoritmo. El enfoque de la programación genética es similar al enfoque conexionista: consiste en encontrar un programa, es decir, la implementación de un algoritmo, que resuelva un problema. En este sentido se parece más a las redes neuronales que a los algoritmos genéticos.

Existen miles de aplicaciones industriales y comerciales de la computación evolutiva, estas son algunas de las más interesantes:

Elaboración de retratos robots de sospechosos: Richard Dawkins, un biólogo evolucionista, incluyó en su libro titulado "*Un relojero ciego*" un programa llamado **BIOMORFOS**, donde se presentaban una serie de formas ligeramente parecidas a insectos, y se podía seleccionar con el mouse aquellas que se asemejaban más a un insecto; estas formas se reproducían y el proceso se repetía por unas cuantas generaciones. El resultado era que en la generación final aparecían formas sorprendentemente parecidas a insectos reales. Inspirados en esto, los investigadores Caldwell y Johnston de la Universidad de Nuevo México usaron los procedimientos policiales habituales para paramétrizar rostros humanos, de manera que podían ser almacenados en un cromosoma. A la víctima de un delito se le presentaban varios rostros, para que seleccionara cuál o cuáles se parecían más al sospechoso, asignándoles una puntuación. Por medio de procedimientos genéticos los cromosomas correspondientes se reproducían, obteniendo un retrato virtual bastante fiel del sospechoso.

Diseño de redes neuronales: En este caso el aprendizaje neuronal se mezcla con el genético para dar una combinación ideal. El diseño de una red neuronal conlleva varias dificultades, los algoritmos genéticos se usan para optimizar, principalmente los pesos iniciales de una red, su tamaño y las constantes de aprendizaje. El hallar los pesos iniciales correctos es esencial ya que el no hacerlo puede provocar que el algoritmo de aprendizaje se quede estacionado en un mínimo local. La optimización de las constantes de aprendizaje también es importante; ya que una constante demasiado grande ocasiona que la red olvide demasiado pronto, y una constante muy pequeña que la red no aprenda en el tiempo necesario. Además el tamaño debe ser el ideal para el tipo de problema, si hay demasiadas neuronas el entrenamiento y la explotación serán muy lentas, si hay muy pocas el porcentaje de acierto puede no ser el apropiado. Para resolver estos problemas se crea una población de redes neuronales, se entrenan, se prueban y se asigna como fitness la exactitud de la clasificación de las muestras de entrada.

Aprendizaje basado en algoritmos genéticos: Es un método singularmente parecido a las redes neuronales, pero en cierto sentido mucho más complicado. Habitualmente se aplica a clasificadores, que deben ser capaces de ordenar correctamente los patrones que se van introduciendo. Para ello tienen un sistema interno de envío de mensajes, que son recogidos por una serie de reglas, que a su vez pueden emitir mensajes. Sobre de las reglas y en función del número de veces que hayan sido útiles se aplica un algoritmo genético. Su representante más notable es el **ANIMAT** de Wilson, un programa artificial que aprende a moverse dentro de un bosque virtual. Actualmente y debido a su mayor facilidad, este tipo de enfoques han sido sustituidos por las redes neuronales.

Con esta información se tiene un panorama general del campo de la informática evolutiva, y las ideas que dieron origen a ella. Existen también por supuesto, los algoritmos genéticos pero considerando que son el tema central de esté trabajo de tesis, se le estudia como el tema central del siguiente capítulo.

CAPITULO II

ALGORITMOS GENÉTICOS

La computación evolutiva es un conjunto de técnicas que basan su funcionamiento en modelar procesos evolutivos fundamentados en la supervivencia de los individuos más aptos de una población. Dentro de dichas técnicas se encuentran los algoritmos genéticos, que son el tema a exponer en este capítulo.

2.1 Breve Historia.

John Holland sería el hombre que daría forma al concepto de algoritmo genético. Fue a principios de los años 60, en la universidad de Michigan en Ann Arbor, donde, dentro del grupo "Lógica de computadoras" sus ideas comenzarían a desarrollarse y dar frutos. Sería además dentro de esta etapa que la lectura del libro escrito por el Biólogo evolucionista, R.A. Fisher, titulado "La teoría genética de la selección natural" influyó en él y sus ideas. De este libro aprendió que la evolución era una forma de adaptación más potente que el simple aprendizaje, y tomó la decisión de aplicar estas ideas para desarrollar programas bien adaptados para un fin determinado.

En la universidad, Holland impartía un curso titulado "Teoría de sistemas adaptativos"; dentro de este curso, y con una participación activa de sus estudiantes, crearía las ideas que más tarde serían los algoritmos genéticos, a los que originalmente Holland llamo "Planes Reproductivos".

Los objetivos de Holland en su investigación serían:

1) Imitar los procesos adaptativos de los sistemas naturales.

2) Diseñar sistemas artificiales (normalmente programas) que utilicen los mecanismos de los sistemas naturales.

Unos 15 años más adelante, David Goldberg, conoció a Holland, y se convirtió en su estudiante. Goldberg era un ingeniero industrial y fue de los primeros en tratar de aplicar los algoritmos genéticos a problemas industriales (diseño de pipelines). Si bien Holland trato de disuadirle, considerando que el problema era demasiado complejo para la aplicación de algoritmos genéticos, Goldberg logro su objetivo escribiendo su programa en una computadora Apple II. Estas y otras aplicaciones convirtieron a los algoritmos genéticos en un campo suficientemente aceptado, y en 1985 celebró su primer conferencia ICGA´85. Dicha conferencia se sigue celebrando bianualmente.

2.2 Definición de algoritmo genético.

David Goldberg define el concepto de algoritmo genético la siguiente manera:

"*Algoritmos de búsqueda basados en el mecanismo de selección natural y genética natural. Combinan la supervivencia de la más apta entre una estructura dada de cadenas con un intercambio aleatorio de información para conformar un algoritmo de búsqueda con algo del talento de la búsqueda humana*".[(6)]

Se puede considerar también que los algoritmos genéticos son métodos sistemáticos para la resolución de problemas de búsqueda y optimización que aplican métodos de la evolución biológica como son la selección basada en la población, la reproducción sexual y la mutación.

Así entonces se puede considerar que los algoritmos genéticos son métodos de optimización, que tratan de hallar el conjunto de X_i, ... , X_j tales que $F(X_i, ... , X_j)$ sea un máximo. En un algoritmos genético, las X_i, ... , X_j se codifican en un cromosoma, (o gen, ya que se consideran equivalentes) que tiene toda la información necesaria para resolver el problema. Después todos los operadores utilizados por el algoritmo genético se aplicarán a una población de cromosomas. Las soluciones codificadas en los cromosomas compiten para ver cuál constituye la mejor solución. El ambiente formado por las demás soluciones ejercerá una presión selectiva sobre la población, de forma que sólo los mejores sobrevivan o leguen su material genético a las siguientes generaciones, del mismo modo que en la evolución de las especies. La diversidad genética se introduce mediante mutaciones y reproducción sexual.

En la naturaleza el factor que hay que optimizar es la supervivencia del individuo y con él la de la especie, eso significa a su vez maximizar diversos factores y minimizar otros. Un algoritmo genético, sin embargo, tiene cómo objetivo habitual optimizar sólo una función, y no diversas funciones relacionadas entre sí simultáneamente. Este tipo de optimización, denominada multimodal, también se suele abordar con un algoritmo genético especializado.

El algoritmo genético trabaja a nivel del genotipo y su operador primario será la cruza sexual ya que modela la evolución a nivel de los individuos; su operador secundario es la mutación. La manera de seleccionar individuos es probabilística y basada en aptitudes.

Ya que la intención del algoritmo genético es combinar el proceso de selección natural y su mecanismo la genética para resolver un problema de optimización; existen cinco elementos primordiales necesarios para modelar un proceso evolutivo en la computadora, los cuales se enuncian a continuación.

1. La representación adecuada para las soluciones del problema (codificación).

[(6)] Goldberg David Genetic algorithms in searchs, Optimization and machine learning. USA Ed Addison-Wesley, 1989.

2. Método de generación de una población inicial de individuos.
3. Un ambiente, normalmente modelado por una función de evaluación la cual comparará a los individuos de acuerdo a su aptitud.
4. Operadores de reproducción que permitan generar nuevos individuos.
5. Valores para los parámetros del algoritmo (probabilidades, cruza, mutación y número de individuos).

Sintetizando el concepto, un algoritmo genético consiste en averiguar de qué parámetros depende el problema, se codifican estos en un cromosoma, y se aplican los métodos de la evolución: selección y reproducción sexual con intercambio de información y alteraciones que generen la diversidad.

Se pueden enunciar ahora los pasos a seguir en la ejecución de un algoritmo genético.

Algoritmo genético.

1. Generar una población inicial aleatoria de n>0 individuos donde cada uno de ellos representa una solución potencial al problema.
2. Seleccionar a los individuos más aptos de acuerdo a los resultados de evaluarlos con una función de aptitud (fitness).
3. Aplicar operadores de reproducción (cruza, mutación, etc) para asegurar el intercambio genético y la creación de una nueva generación de individuos.
4. Iterar hasta alcanzar una condición de paro del algoritmo previamente establecida.

Los puntos importantes a considerar al diseñar un algoritmo genético serán la codificación de los cromosomas. Los operadores genéticos (principalmente la selección, el crucé y la mutación) serán comentados más ampliamente en las siguientes secciones. Los siguientes parámetros a fijar para cada algoritmo genético y que tienen vital importancia son:

- El tamaño de la población: Debe ser suficiente para garantizar la diversidad de las soluciones.
- La condición de terminación: Lo más habitual es que la condición de terminación sea un criterio de convergencia del algoritmo o un número prefijado de generaciones.

2.3 Codificación.

Los algoritmos genéticos requieren que cada solución sea codificada en un cromosoma, el cual estará conformado por una estructura de datos que representará cada individuo de la población, normalmente en forma de arreglo; donde cada posición es conocida como gen, el cual codifica el valor de un solo parámetro de la solución del problema. El valor que puede tomar cada gen se denomina como alelo.

Considere el siguiente ejemplo consistente en 14 genes representados por dos diferentes alelos en un cromosoma:

Alelo: 0, 1

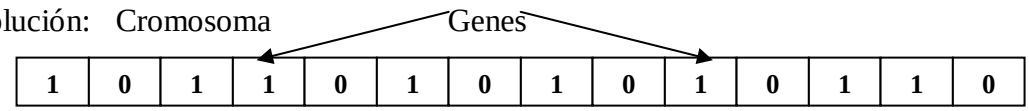


Figura 2.1 Codificación

El conjunto total de cromosomas forma la población sobre la que trabaja el algoritmo genético, en casos específicos esta población estaría dividida en subpoblaciones. De esta manera se modela la especialización, ya que sólo se cruzan individuos de la misma subpoblación. Sin embargo podría permitirse que algunos individuos transfieran sus genes de una subpoblación a otra. A ésto se le conoce como migración. Cuando un individuo puede reproducirse con otro sin importar en que subpoblación esté y que su elección ello sólo dependa de su aptitud se dice que la población es panmítica.

Para poder trabajar con estos cromosomas en una computadora, es necesario codificarlos en una cadena, es decir una ristra de símbolos (números o letras) que generalmente estará compuesta de ceros y unos.

Las representaciones más utilizadas son:

Binaria: Utilizando 0s y 1s para representar cada parámetro.

Ejemplo:

Supóngase que se quiere hallar el máximo de la función $f(x)=1-X^2$, una parábola invertida con el máximo en X=1. En este caso, el único parámetro del problema será la variable X; la optimización consiste en hallar una X tal que f(x) sea máximo. Se crea una población de cromosomas, cada uno de los cuales contiene una codificación binaria del parámetro X. Se procede de la siguiente manera, los valores enteros elegidos en un intervalo [–100, 100] serán convertidos a sus valores binarios y evaluados en la función.

Valor Binario	Valor Entero	Evaluación f(x)
10101	21	-440
10011	19	-360
1010110	-86	-7395
111010	-58	-3363
1	1	0

Figura 2.2 Codificación binaria

Comúnmente los números binarios son utilizados con cadenas de longitud fija, donde en cada posición se indica el estado del cromosoma según aparezca un cero o un uno en cada posición.

Binaria con código Gray: El código Gray es un tipo de codificación basado en el sistema binario pero de una construcción muy distinta a la de los demás códigos. Su principal característica es que dos números sucesivos cualesquiera solo varían en un bit. Esto se consigue mediante un procedimiento poco riguroso consistente en:

- Se escribe en una columna los dígitos 0 y 1.
- Se toma una línea imaginaria en la base de la columna.
- Se reproduce la columna bajo la línea como si de un espejo se tratase.
- Se rellena con ceros la parte superior y con unos la inferior.

Por lo tanto para un código Gray de n bits se toma el correspondiente Gray de n-1 bits, se le aplica simetría y se rellena su parte superior con 0s y la parte inferior con 1s. Esta codificación no tiene nada que ver con un sistema de cuantificación; en efecto los términos 000, 101, etc no denotan un valor matemático real sino uno de los X valores que puede tomar una variable. Por lo tanto se trata de hallar, a partir de una variable que puede tomar X valores, un n suficiente como para que $2^n > X$ y ordenar estos estados de la variable de acuerdo a las normas del código Gray de cambio entre dos estados sucesivos.

DECIMAL	BINARIO	GRAY
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Figura 2.3 El código Gray sólo cambia un bit en cada posición.

Esta codificación es muy usada en el diseño de circuitos combinacionales, ya que eventualmente sólo realizar un cambio es muy importante en dispositivos mecánicos ya que el error medio en la transmisión es menor con menos cambios, como ocurre cuando se emplea el código Gray.

Entera: Utiliza los números enteros positivos para codificar las diversas posibilidades de solución de un problema.

Real: Codifica en forma de número real cada posición del cromosoma, es similar a la entera pero amplía la gama de posibilidades de codificación incluyendo los números reales.

Expresiones S en LISP (árboles): Este tipo de codificación dio origen a la programación genética propuesta por Kosa, se considera que todo lo que puede ser programado en LISP se puede representar de esta manera.

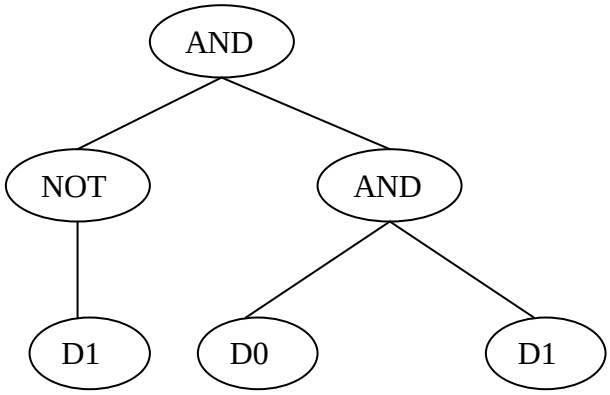


Figura 2.4 Árbol LISP

Listas binarias de longitud variable: En estos casos el código binario es utilizado como un número, de forma que la longitud de la cadena puede variar según el número que representa o la cantidad de parámetros que codifica, la decisión de utilizar cadenas dinámicas o emplear cadenas de longitud fija esta determinada por las características particulares de cada problema.

Híbridos: Son diversas combinaciones utilizadas para codificar las soluciones, generalmente de acuerdo a las características especiales de cada problema.

La cardinalidad de un alfabeto índica el número de elementos en él. Existen otras codificaciones posibles, usando alfabetos de diferente cardinalidad; sin embargo, en uno de los resultados fundamentales en la teoría de los algoritmos genéticos, llamado teorema de los esquemas, se afirma que la codificación óptima, es decir, aquella sobre la que los algoritmos genéticos funcionan mejor, es aquella que tiene un alfabeto de cardinalidad dos.

La mayoría de las veces, una codificación correcta es la clave de una buena resolución del problema. Generalmente la regla heurística que se utiliza es la llamada regla de los bloques de construcción, es decir, parámetros relacionados entre sí deben estar cercanos dentro del cromosoma.

En todo caso, se puede ser bastante creativo con la codificación del problema, teniendo en cuenta la siguiente regla: variables del problema que estén juntas en el espacio-problema, deben de estar juntas en la codificación. Esto puede llevar en algunos casos a cromosomas bidimensionales o tridimensionales, o con relaciones entre genes que no sean puramente lineales. En algunos casos, cuando no se conoce de antemano el

número de variables del problema, caben dos opciones: codificar también el número de variables, o bien, lo cual es mucho más natural, crear un cromosoma que pueda variar de longitud. Para ello, claro está, se necesitan operadores genéticos que alteren la longitud del cromosoma.

2.4 Evaluación y Selección.

La selección de los individuos es de suma importancia en un algoritmo evolutivo ya que es la guía de la búsqueda hacia una buena solución. Durante la evaluación se decodifica el gen, convirtiéndolo en una serie de parámetros del problema, se encuentra la solución del problema a partir de esos parámetros, y se le da una puntuación a la solución en función de lo cerca que se encuentre de la mejor solución. A esta puntuación se le llama fitness.

El fitness determina siempre los cromosomas que van a reproducir, y aquellos que se eliminaran, pero existen varias formas de considerar el criterio para seleccionar la población de la siguiente generación.

- Selección proporcional: Se eligen individuos de acuerdo a su contribución de aptitud con respecto al total de su población. Por ejemplo:

    1. Ruleta.
    2. Sobrante estocástico.
        - Con reemplazo.
        - Sin reemplazo.
    3. Universal estocástica.
    4. Muestreo determinístico.

- Selección mediante torneo: Propuesta por Wetzel. Se basa en comparaciones directas de aptitud entre individuos. Existen dos tipos:

    1. Determinística.
    2. Probabilística.

- Selección de estados uniformes: Propuesta por Whitley, utilizada en algoritmos genéticos no generacionales y aprendizaje incremental. Útil cuando los individuos resuelven el problema de manera colectiva.

Uno de los criterio de selección más usados es el de Ruleta: En esté método se crea un pool genético formado por cromosomas de la generación actual, en una cantidad proporcional a su fitness. Si la proporción hace que un individuo domine la población, se le aplica alguna operación de escalado. Dentro de este pool, se escogen parejas aleatorias de cromosomas y se emparejan, sin importar que sean del mismo progenitor. Dentro de este sistema existen algunas variaciones, una de ellas es el "*elitismo*", que

consiste en incluir el mejor (o los mejores) representantes de la generación actual en la siguiente de manera automática.

Otra técnica a considerar es el llamado Estado Estacionario: en este esquema se mantiene un porcentaje de la población para la siguiente generación. Se coloca toda la población por orden de fitness, y los M menos dignos son eliminados y sustituidos por la descendencia de alguno de los N mejores con algún otro individuo de la población. A este esquema se le aplican también algunas variantes, por ejemplo crear la descendencia de uno de los mejores y sustituir con estos al más parecido entre los perdedores. Esto se denomina crowding y fue introducido por DeJong. En realidad para este esquema se escoge un *crowding factor* (CF), cuando nace una nueva criatura se seleccionan CF individuos de una población, y se elimina al más parecido a la nueva criatura.

Los operadores de reproducción también llamados Operadores Genéticos son aquellos que modifican la manera en que se transmite la información genética de padres a hijos, los principales son el cruce y la mutación. Existen además dos subprocesos relacionados con los operadores genéticos.

Uno de ellos es el llamado de "*explotación*", el cual consiste en utilizar la información obtenida anteriormente para decidir cuál es el más conveniente utilizar continuación mediante movimientos finos. El operador de cruce permite explotar una zona prometedora del espacio de búsqueda del problema y con ello encontrar óptimos locales en ese lugar.

El otro proceso es llamado "*exploración*", el cual consiste en encontrar zonas prometedoras del espacio de búsqueda e impedir el quedar atrapado en óptimos locales. La mutación permite dar saltos significativos en el espacio para lograr este cometido. El cruce y la mutación serán el tema de las dos siguientes secciones.

2.5 Cruce.

También llamado en Inglés Crossover, es el principal operador genético, hasta el punto de que se puede decir que un algoritmo, no es un algoritmo genético si no tiene cruce, y sin embargo puede serlo sin tener mutación, como afirma Holland; el teorema de los esquemas confía en él para hallar la mejor solución a un problema. Básicamente consiste en el intercambio de material genético entre dos cromosomas (en ocasiones pueden ser más, como el operador "*orgía*" propuesto por Eiben).

Para aplicar el cruce se escogen aleatoriamente dos miembros de la población; no existe problema en emparejar dos descendientes de los mismos padres, ya que ello garantiza la perpetuación de un individuo con buena puntuación (además algo parecido ocurre en la realidad, ya que es una practica utilizada en la cría de ganado, llamada inbreeding y está destinada a potenciar ciertas características frente a otras). Sin embargo si se debe considerar que si esto sucede demasiado a menudo, puede crear problemas ya que toda la población puede ser dominada por los descendientes de algún gen, que además puede tener caracteres no deseados. Esto es lo que en otros métodos de optimización se

denomina “atraque en un mínimo local”, y es uno de los principales problemas con los que se tiene que lidiar al usar algoritmos genéticos.

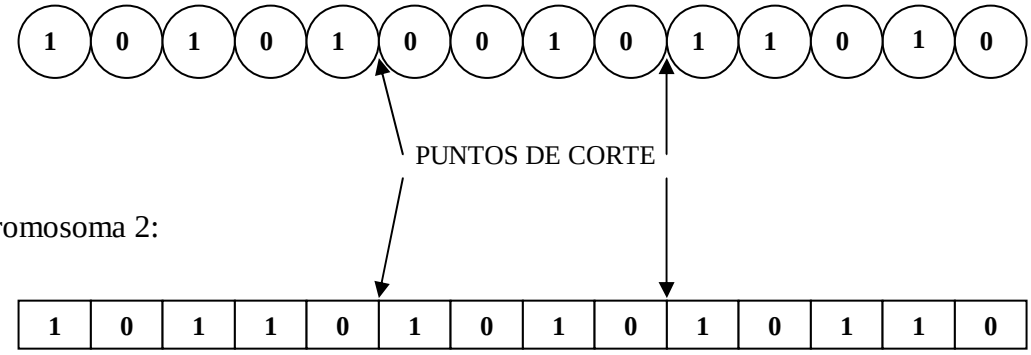
En cuanto al teorema de los esquemas, se basa en la noción de “*bloques de construcción*”. Se consideran bloques a las secciones de cromosomas, una buena respuesta estará constituida por buenos bloques, igual que una buena máquina está construida con buenas piezas. El cruce es el encargado de mezclar bloques buenos que se encuentran en los diversos progenitores, y que serán los que den a los individuos una buena puntuación. La presión selectiva se encarga de que sólo los buenos bloques se perpetúen, y poco a poco se vaya formando una buena solución. En conclusión el teorema de los esquemas nos dice que la cantidad de buenos bloques se va incrementando con el tiempo de ejecución, y es el resultado teórico más importante en algoritmos genéticos.

El intercambio genético se puede llevar a cabo de muchas formas, pero hay tres grupos principales:

Cruce de n-puntos: En este cruce los dos cromosomas se cortan por n puntos, y el material genético situado entre ellos se intercambia. El más común es el cruce de uno o dos puntos.

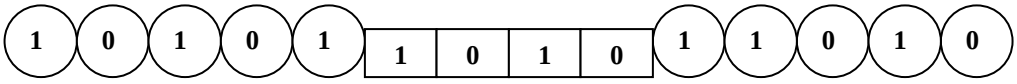
El siguiente es un ejemplo de cruce de dos puntos, se tienen dos cromosomas, se eligen dos puntos de corte y se combinan los segmentos separados para obtener dos nuevos individuos.

Cromosoma 1:



DOS NUEVOS INDIVIDUOS DESPUÉS DEL CRUCE

Nuevo cromosoma 1:



Nuevo cromosoma 2:

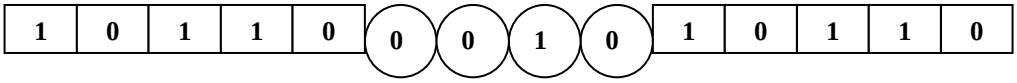
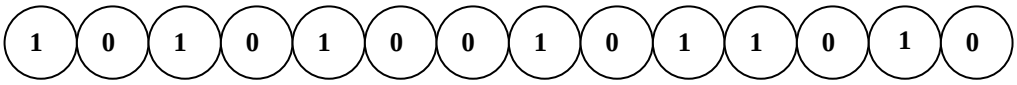


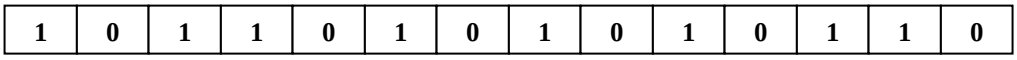
Figura 2.5 CRUCE DE DOS PUNTOS

Ahora se estudia un ejemplo de cruce multipunto, donde se realizan múltiples cortes y combinaciones entre los dos cromosomas.

Cromosoma 1:

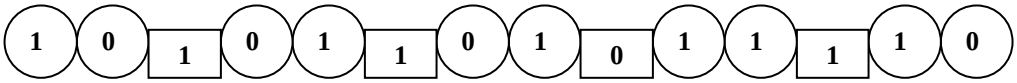


Cromosoma 2:



DESPUES DEL CRUCE

Nuevo cromosoma 1:



Nuevo cromosoma 2:

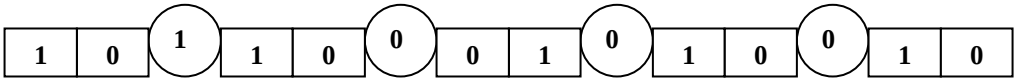


Figura 2.6 CRUCE MULTIPUNTO

Cruce uniforme: Se genera un patrón aleatorio de 1s y 0s, y se intercambian los bits de los dos cromosomas que coincidan, donde exista un uno en el patrón. O bien se genera un número aleatorio para cada bit, y si supera una determinada probabilidad se intercambia ese bit entre los dos cromosomas.

Cruce especializado: En algunos problemas, el aplicar aleatoriamente el cruce da lugar a cromosomas que codifican soluciones no válidas; en este caso hay que aplicar el cruce de forma que genere siempre soluciones válidas, las especialidades vienen dadas por las características de los problemas, como en la programación genética de Koza.

2.6 Mutación

En la evolución, una mutación es un suceso bastante poco común (sucede aproximadamente una de cada mil creaciones), en la mayoría de los casos las mutaciones son letales, pero en promedio contribuyen a la diversidad genética de la especie. En un algoritmo genético tendrán el mismo papel y la misma frecuencia (es decir, muy baja).

Una vez establecida la frecuencia de mutación, por ejemplo uno por cada mil, se examina cada bit de cada cadena cuando se crea la nueva criatura a partir de sus padres (normalmente se hace de forma simultánea al cruce). Si un número generado aleatoriamente está por debajo de la probabilidad, se cambiará el bit (es decir, de 0 a 1 o viceversa). En caso contrario se conserva como está. Dependiendo del número de individuos que existan y del número de bits por individuo, puede resultar que las mutaciones sean extremadamente raras en una sola generación.

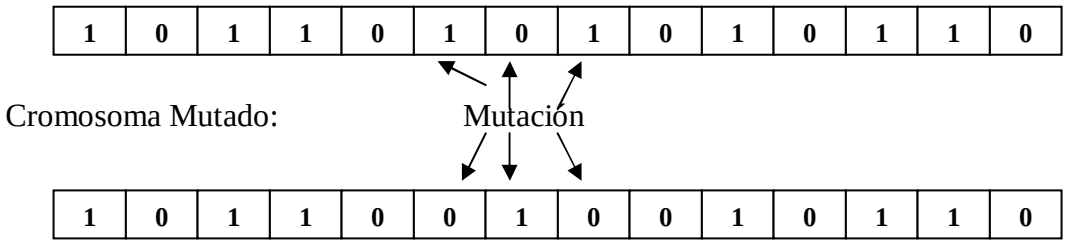


Figura 2.7 Mutación

Es necesario señalar que no conviene abusar de la mutación. Es cierto que es un mecanismo generador de diversidad, y por lo tanto una solución cuando un algoritmo genético está estancado, pero también es cierto que reduce el algoritmo genético a una búsqueda aleatoria. Siempre es más conveniente usar otros mecanismos de generación de diversidad, como aumentar el tamaño de la población, o garantizar la aleatoriedad de la población inicial. Esta operación, junto con el cruce y el método de selección de ruleta, constituyen un "*algoritmo genético simple*", SGA, introducido por Goldberg en su libro "*Algorithms in searchs, Optimization and machine learning*".

2.7 Otros operadores.

Existen diferentes problemas, con características muy distintas que pueden generar la necesidad de uso de diferentes operadores. Por ejemplo, cuando se conoce de antemano la cantidad de parámetros que hay en un problema se puede trabajar con cromosomas de longitud fija, pero en un problema de clasificación, donde dado un vector de entrada, se desea agruparlo en una serie de clases, se podría ignorar cuantas clases existen. O en el diseño de redes neuronales, puede que no se sepa (de hecho, nunca se sabe) cuántas neuronas se van a necesitar; en un perceptrón existen reglas que indican cuantas

neuronas se deben utilizar en la capa oculta; pero en un problema determinado puede que no haya ninguna regla heurística aplicable; se tendrán que utilizar los algoritmos genéticos para encontrar el número óptimo de neuronas. En este caso, si se utiliza una codificación "*fregona"* (aquella donde se ponen juntos todos los pesos que salgan de la misma neurona de la capa oculta), se necesitara un locus para cada neurona de capa oculta, y el número de locus variará dependiendo del número de neuronas de la capa oculta.

En estos casos, se necesitan dos operadores más: *añadir* y *eliminar*. Se utilizan para añadir un locus, o eliminar un locus del cromosoma. La forma más habitual de añadir un locus es duplicar uno ya existente, el cual sufre mutación y se añade al lado anterior. En este caso los operadores del algoritmo genético simple (selección, mutación y cruce) funcionarán de la forma habitual, salvo, claro está, que sólo se hará cruce en la zona del cromosoma de menor longitud.

Estos operadores permiten, además, crear un algoritmo genético de dos niveles: a nivel de cromosoma y a nivel de alelo. Supóngase que en un problema de clasificación, existe un alelo por clase, se puede asignar a cada alelo una puntuación en función del número de muestras que haya clasificado correctamente. Al aplicar estos operadores, se duplicarán los alelos con mayor puntuación, y se eliminarán aquellos que hayan obtenido una puntuación menor o nula.

Se tiene como ejemplo un problema que consiste en clasificar los puntos del cuadrado de 10 X 10 en dos clases, 1 y 2, que no son linealmente separables. Inicialmente no se sabe cuantos vectores son necesarios para clasificar estas clases. El algoritmo genético debe ser capaz de hallar un número óptimo de vectores, a cada uno de los cuales se asigna una etiqueta de clase, tales que el error se hace mínimo, en este caso cuatro vectores para la primera clase y cinco para la segunda. Cada cromosoma estará compuesto por un diccionario o conjunto de vectores, cada uno de los cuales tiene asignada una etiqueta de clase.

Operadores de nicho ecológico: Estos operadores están encaminados a mantener la diversidad genética de la población, de forma que los cromosomas similares sustituyan sólo a cromosomas similares, y son especialmente útiles en problemas con muchas soluciones; un algoritmo genético con este tipo de operadores es capaz de hallar todos los máximos, dedicándose cada especie a un máximo. Una de las formas de llevar a cabo esto es con el ya mencionado crowding, otra forma es introducir una función de comparación, que indica cuán similar es un cromosoma al resto de la población; la puntuación de cada individuo se dividirá por esta función de comparación, de forma que se facilita la diversidad genética y la aparición de individuos diferentes.

También se pueden restringir los emparejamientos, por ejemplo entre aquellos cromosomas que sean similares, para evitar las malas consecuencias del inbreeding que suelen aparecer en poblaciones pequeñas, estos periodos se intercalan con otros periodos en los cuales el emparejamiento es libre.

Operadores especializados: En una serie de problemas hay que restringir las nuevas soluciones generadas por los operadores genéticos, pues no todas las soluciones generadas van a ser válidas, sobre todo en los problemas con restricciones. Para este fin se cuenta con los siguientes operadores especiales que mantienen la estructura del problema, otros son simplemente generadores de diversidad:

- **Zap**: En vez de cambiar un solo bit de un cromosoma, cambia un gen completo de un cromosoma.
- **Creep**: Este operador aumenta o disminuye en 1 el valor de un gen; sirve para cambiar suavemente y de forma controlada los valores de los genes.
- **Transposición**: Similar al cruce y a la recombinación genética, pero dentro de un solo cromosoma; dos genes intercambian sus valores, sin afectar al resto del cromosoma.

Otro aspecto que requiere especial importancia al aplicar algoritmos genéticos es decidir con que frecuencia se va a aplicar cada uno de los operadores. La frecuencia de aplicación de cada operador estará en función del problema, teniendo en cuenta los efectos de cada operador, tendrá que aplicarse con cierta frecuencia o no. Generalmente, la mutación y otros operadores que generen diversidad se suelen aplicar con poca frecuencia; la recombinación se suele aplicar con frecuencia alta.
En general la frecuencia de los operadores no varía durante la ejecución del algoritmo, pero se debe tener en cuenta que cada operador es más efectivo en un momento determinado de la ejecución. Por ejemplo al principio, en la fase denominada de exploración, los más eficaces son la mutación y la recombinación; posteriormente cuando la población ha convergido en parte, la recombinación no es útil, pues se está trabajando con individuos bastante similares y es poca la información que se intercambia. Sin embargo, si se produce un estancamiento, la mutación puede ser útil, en todo caso se pueden utilizar operadores especializados.

2.8 El Zen y los algoritmos genéticos.

Este es el título de un articulo publicado por Goldberg en la conferencia sobre algoritmos genéticos celebrada en 1989 (ICGA 89), en donde da una serie de consejos para la aplicación debida de los algoritmos genéticos y avisa a aquellos que se quieren apartar de la ortodoxia. En resumen estos son los consejos:

DEJA QUE LA NATURALEZA SEA TU GUÍA: Dado que la mayoría de los problemas a los que se van a aplicar los algoritmos genéticos son de naturaleza no lineal, lo mejor es actuar como lo haría la naturaleza, aunque intuitivamente pueda parecer la forma menos acertada. *"Si queremos desarrollar sistemas no lineales que busquen y aprendan, mejor que comencemos (como mínimo) imitando a sistemas que funcionan"*. Y estos sistemas se hallan en la naturaleza.

CUIDADO CON EL ASALTO FRONTAL: A veces se plantea el problema de pérdida de diversidad genética en una población de cromosomas. Hay dos formas de resolver este problema: aumentar el ritmo de mutación, lo cual equivale a convertir un

algoritmo genético en un algoritmo de búsqueda aleatoria, o bien introducir mecanismos como el sharing, medio por el cual el fitness de un individuo se divide entre el número de individuos similares a él. Este segundo método, más parecido al funcionamiento de la naturaleza, en la cual cada individuo, por bueno que sea, tiene que compartir recursos con aquellos que hayan resuelto el problema de la misma forma, funciona mucho mejor.

RESPETA LA CRIBA DE ESQUEMAS: Para realizar esto, lo ideal es utilizar alfabetos de baja cardinalidad (es decir, con pocas letras) como el binario.

NO TE FÍES DE LA AUTORIDAD CENTRAL: La Naturaleza actúa de forma distribuida, por lo tanto se debe de minimizar la necesidad de operadores que "*vean*" completa a la población. Ello permite, además, una fácil paralelización del algoritmo genético. Por ejemplo, en vez de comparar el fitness de un individuo con todos los demás, se pueden comparar sólo con los vecinos, es decir aquellos que estén de alguna forma situados cerca de él.

2.9 Aplicaciones

Para aplicar algoritmos genéticos se debe considerar que en el problema a resolver, la meta ha de poder ser observada en grados cualitativamente comparables. Por otra parte, las principales dificultades a la hora de implementar un algoritmo genético son:

- Definir una estructura de datos que pueda contener patrones que representen, la solución óptima buscada (desconocida) y todas las posibles alternativas de aproximaciones a la solución.
- Definir un tipo de patrón tal que si un patrón es seleccionado positivamente, que esto no sea debido a la interacción de los distintos segmentos del patrón, sino que existan segmentos que por sí solos provocan una selección positiva.
- Definir una función de evaluación que seleccione los mejores individuos.

Además las condiciones que debe cumplir un problema para ser abordable con algoritmos genéticos son:

- El espacio de búsqueda debe ser acotado.
- Debe existir un procedimiento relativamente rápido que asigne un grado de utilidad a cada solución propuesta, de forma que este grado de utilidad asignado corresponda, o bien directamente con la calidad de la solución en cuanto al problema a resolver, o bien con un valor de calidad relativo al resto de la población que permita obtener en el futuro mejores soluciones.
- Debe existir un método de codificación de soluciones que admita la posibilidad de que los cruzamientos combinen las características positivas de ambos progenitores. Este método debe permitir también aplicar algún mecanismo de mutación que sea capaz de obtener, tanto soluciones muy dispares respecto de la solución sin mutar, como muy parecidas a ésta.

Los campos de aplicación de los algoritmos genéticos son tan amplios como la imaginación humana, sin embargo de acuerdo con las condiciones antes expuestas se pueden clasificar las aplicaciones de la siguiente manera:

- Decisión y Estrategia
  - Toma de decisiones financieras (presupuestos)
  - Búsqueda de reglas en juegos.
  - Experimentación de alternativas de marketing
  - Gestión de franquicias.
- Diseño y parametrización
  - Diseño de pistas de circuitos integrados VLSI
  - Parametrización de Sistemas (configuraciones de equipos hardware)
  - Diseño de redes (telecomunicaciones, carreteras) Ubicación de nodos
- Planificación y asignación de recursos
  - Asignación del orden de N procesos en M CPU
  - Ordenación (ordenar cajas en un almacén)
  - Asignación de horarios de clase o turnos de trabajo en un hotel
  - Resolución de sistemas de ecuaciones no lineales
  - Enrutamiento
  - El problema del viajante de comercio (TSP)
- Predicción
  - Selección de combinaciones de métodos de realización de series temporales

Con esta información se considera que se cuenta con los conocimientos suficientes para entender la solución de problemas por medio de algoritmos genéticos. En el siguiente capítulo se estudia más a fondo el problema del agente viajero, ya que en él se enfocara el algoritmo genético a desarrollar.

CAPÍTULO III

EL PROBLEMA DEL AGENTE VIAJERO

Cuando la teoría de la computación se desarrolló, lo más lógico fue preguntarse acerca de la relativa dificultad computacional de las funciones computables. Este es el planteamiento de la Complejidad Algorítmica. Rabin en 1960 se preguntó: ¿Qué quiere decir que f sea más difícil de computar que g? Rabin ideó una axiomática que fue la base para el desarrollo de la complejidad abstracta de Blum y otros.

En 1965 Harmantis y Stearns escribieron el artículo "*On the computacional complexity of algorithms*", gracias a este artículo se le dio nombre a este cuerpo de conocimiento. Se fundamentó así la medida de complejidad definida como el tiempo de computación sobre una máquina de Turing multicinta, y se demostraron los teoremas de jerarquía.

En el mismo año de 1965 Cobham escribió "*The intrinsic computacional difficulty of functions*", en donde enfatizó la palabra intrínseco, porque estaba interesado en una teoría independiente de las máquinas. También definió y caracterizó la importante clase de funciones llamadas λ es decir las funciones de números naturales que son computables en tiempo acotado por un polinomio cuyo argumento es la longitud decimal de la entrada. [(1)]

En computación cuando el tiempo de ejecución de un algoritmo (mediante el cual se obtiene una solución al problema) es menor que un cierto valor calculado a partir del número de variables implicadas (generalmente variables de entrada) usando una fórmula polinómica, se dice que dicho problema se puede resolver en un "Tiempo polinómico" o también llamado "Tiempo Polinomial".

En teoría de la complejidad, la clase de complejidad de los problemas de decisión que pueden ser resueltos en tiempo polinómico calculado a partir de la entrada por una máquina de Turing determinista es llamada P. Cuando se trata de una máquina de Turing no-determinista, la clase se llama NP. Una de las preguntas abiertas más importantes en la actualidad es descubrir si estas clases son diferentes o no. El Clay Mathematics Institute ofrece un millón de dólares a quien sea capaz de responder esa pregunta.

Antes de hablar de problemas NP, es necesario introducir la noción de no determinismo. Un algoritmo no determinista es aquél que entre sus operaciones permitidas incluye una primitiva especial llamada *elección nd* (no determinista). En problemas de decisión, dada una entrada x, el algoritmo no determinista efectuará una secuencia de pasos (deterministas) entrelazados con otros donde se use la operación de elección *nd*, y al final decidirá si se acepta x. Un algoritmo no determinista reconoce un

---

(1) Castro Peña Juan Complejidad Algorítmica http://decsai.ugr.es/~castro/CA/node24.html

lenguaje L si, para una entrada x, es posible convertir toda elección *nd* encontrada durante la ejecución en una elección determinista tal que la salida del algoritmo sea aceptar x si y solo si $x \in L$. En otras palabras, el algoritmo debe proporcionar por lo menos una secuencia posible de pasos para aceptar datos pertenecientes a L, y no debe proporcionarla para entradas que no pertenezcan a L.

Por ejemplo, considérese el problema de decidir si un grafo G = (V, A) tiene un *ajuste perfecto.* (Un ajuste se define sobre un grafo no dirigido G como un subconjunto de sus aristas tales que, tomadas de dos en dos, no tienen ningún vértice común; se dice que un ajuste es perfecto si todos los vértices de G pertenecen a él). El algoritmo utilizado mantiene un conjunto de aristas M inicialmente vacío. Se van examinando todas las aristas de G usando una elección nd para decidir si incluir cada arista A en el conjunto M. Tras examinar todas ellas, se comprueba si M es un ajuste perfecto, lo cual puede hacerse en tiempo lineal, ya que hay que determinar si M contiene exactamente V/2 aristas y si todo vértice es exactamente incidente a una arista de M. La salida del algoritmo será "sí" cuando M sea un ajuste perfecto y "no", en otro caso. Se trata de un algoritmo no determinista correcto, ya que (1) si existe un ajuste perfecto, habrá una secuencia de decisiones que permitirán formar M, y (2) el algoritmo responde afirmativamente, si se comprueba la existencia de un ajuste perfecto.

Por tanto la complejidad se ha dividido en dos clases principales; la clase P y la NP. La clase P corresponde a los problemas cuyos algoritmos de solución son de complejidad en tiempo polinomial; y los NP son los problemas cuya solución hasta la fecha no han podido ser resueltos de manera exacta por medio de algoritmos deterministas eficientes, pero que pueden ser resueltos por algoritmos no deterministas y cuya solución son de complejidad en tiempo polinomial.[(2)] Así se tiene que:

1.- P denota la colección de todos los problemas de decisión los cuales tienen algoritmos determinísticos en tiempo polinomial. [(3)]

Se ha demostrado que tres problemas importantes están en la clase P. El primero es la programación lineal, demostrado por Khachian en 1979. El segundo es determinar si dos grafos de grado máximo d son isomorfos, demostrado por Lux en 1980. El tercero es la factorización de polinomios con coeficientes racionales, que fue demostrado por Lenstra y Lobas en 1982 para polinomios de una variable; la generalización para polinomios de cualquier número fijo de variables fue demostrada por Kaltofen en 1982.[(4)]

2.- NP Polinómico no determinista (Non-Deterministic Polynomial-time) denota la colección de todos los problemas de decisión los cuales tienen algoritmos de solución no determinísticos en tiempo polinomial. [(5)]

[(2)] Edgard Reingold Combinational Algorithms Theory and Practice.
[(3)] Stinson D R An Introduction to de Desing and Análisis of Algorithms 1987.
[(4)] Castro Peña Juan Complejidad Algorítmica http://decsai.ugr.es/~castro/CA/node24.html
[(5)] Stinson D R An Introduction to de Desing and Análisis of Algorithms 1987.

Esto es algoritmos no determinísticos en los cuales hay siempre un camino computacional exitoso que requiere tiempo polinomial en la longitud de la cadena de entrada. (6)

El desarrollo más importante de la complejidad algorítmica es la teoría de la NP-Completitud, por lo que la clase NP consta de todos los conjuntos reconocibles en tiempo polinomial por una máquina de Turing no determinística. En 1962 Bennet bajo el nombre de "relaciones rudimentarias positivas extendidas" definió un equivalente a NP utilizando cuantificadores lógicos en lugar de máquinas de computación. En 1972 Karp le puso el nombre actual de clase NP y Cook introdujo el concepto de NP-Completo y explicó que el problema del subgrafo es NP-Completo. Un año más tarde Karp encuentra 21 problemas NP-Completos, lo que demostraba la importancia de la materia.

Dada su importancia, se han hecho muchos esfuerzos para encontrar algoritmos que resuelvan algún problema de NP en tiempo polinómico. Sin embargo, pareciera que para algunos problemas de NP (los del conjunto NP-completo) no es posible encontrar un algoritmo mejor que simplemente realizar una búsqueda exhaustiva.

En su artículo "Primes in P" publicado en el 2002, Manindra Agrawal junto con sus estudiantes encontró un algoritmo que trabaja en tiempo polinómico para el problema de saber si un número es primo. Anteriormente se sabía que ese problema estaba en NP, si bien no en NP-Completo, ahora se sabe que también está en P.

3. Un problema es NP-Duro si todo problema en NP se puede transformar polinomialmente a él. Es decir cualquier problema de decisión, pertenezca o no a los problemas NP, el cual pueda ser transformado a un problema NP-Completo tendrá la propiedad de que no podrá ser resuelto en tiempo polinomial a menos que P=NP. Se puede entonces decir que dicho problema es al menos tan difícil como uno NP-Completo, razón por la que a los problemas con estas características suele llamárseles SAT además de NP-Duros. SAT (Problema de satisfactibilidad) es considerado como el problema canónico de los problemas en NP que incluye a una gran cantidad de problemas de todo tipo en Informática.

El problema de la suma de subconjuntos es un ejemplo de problema NP-Duro y se define como sigue: Dado un conjunto S de enteros, ¿Existe un subconjunto no vacío de S cuyos elementos sumen cero?

4. Un problema es NP-Completo si es duro y es NP. Mas específicamente, tenemos una amplia variedad de problemas de tipo NP, de los cuales destacan algunos de ellos de extrema complejidad. Gráficamente se dice que algunos problemas se hallan en la "frontera externa" de la clase NP. Son problemas NP, y son los peores problemas posibles de la clase NP. Estos problemas se caracterizan por ser todos "iguales" en el sentido de que si se descubriera una solución P para alguno de ellos, esta solución sería

(6) Edgard Reingold Combinational Algorithms Theory and Practice.

fácilmente aplicable a todos ellos. Actualmente hay un premio de prestigio equivalente al Nóbel reservado para el que descubra semejante solución, y se duda seriamente que alguien lo consiga; si se descubriera una solución para los problemas NP-Completos, está sería aplicable a todos los problemas NP y por tanto, la clase NP desaparecería del mundo científico al carecerse de problemas de ese tipo. Realmente, tras años de búsqueda exhaustiva de dicha solución, es un hecho ampliamente aceptado que no debe existir, aunque nadie ha demostrado, todavía, la imposibilidad de su existencia.

Un problema NP-Completo tiene la característica de que todo problema en NP se reduce polinomialmente a él. Por lo tanto se concluye que es importante resolver problemas NP-Completos, porque si alguien puede resolver un problema NP-Completo en tiempo polinomial se podrán resolver todos los problemas NP-Completos en tiempo polinomial.[(7)]

Por lo que dada la importancia de los problemas NP-Completos y sabiendo que la solución de un problema considerado como NP-completo, repercute en todos los demás problemas del mismo tipo, se selecciona para su estudio y ejemplificación de los algoritmos genéticos el problema clásico NP-Completo: El problema del Agente Viajero.

3.1 Definición del problema.

Los matemáticos se han divertido mucho con problemas muy difíciles que tratan como acertijos, el problema del agente viajero es uno de ellos. La primer noticia que se tiene del problema del agente viajero fue en 1831 en Alemania, donde un libro fue publicado y titulado como “Der Handlungsreisende” o bien “El Agente Viajero”, donde se hacia la pregunta ¿Cómo debe de ser un agente viajero y qué debe hacer para vender más y ser exitoso en su negocio? Respondiendo a esta pregunta con un programa de recorridos para poder cubrir tantas localidades como fuera posible sin visitar una localidad dos veces.

El problema del agente viajero consiste en que: Un viajero debe visitar cada ciudad en su territorio exactamente una vez y debe regresar al punto de partida. Se le da el costo de viajar entre todos los pares de ciudades. Debe plantear su itinerario considerando que debe visitar cada ciudad exactamente una vez y el costo total de su viaje debe ser el mínimo. Por lo que en términos generales el problema es encontrar el costo mínimo del circuito de longitud n. Formalmente el problema se define como:

Sean n ciudades de un territorio. La distancia entre cada ciudad viene dada por la matriz d de n X n, donde d[x,y] representa la distancia que hay entre la ciudad X y la ciudad Y. El objetivo es encontrar una ruta que, comenzando y terminando en una ciudad concreta, pase una sola vez por cada una de las ciudades y minimice la distancia recorrida por el viajante. Es decir, encontrar una permutación:

[(7)] Wolf H S Algorithms and Complexity 1986.

$P = \{c_0, c_1, ..., c_{n-1}\}$ tal que $dp = \sum_{i=1}^{m-1} d\ (C_i, C_i)$ sea mínimo. [8]

Este problema visto desde la óptica de grafos es: un grafo a cuyas aristas se les ha asignado un peso entre los vértices que representan las ciudades que debe visitar el agente viajero que pueden representar kilometraje, costo, tiempo de computadora o alguna otra cantidad que se quiera minimizar. El objetivo es encontrar la ruta mínima que pase por cada ciudad exactamente una vez y regrese a la ciudad inicial. La meta es encontrar un circuito hamiltoniano que minimice la suma de los pesos de las aristas. Un buen algoritmo que resolviera este problema también solucionaría el de encontrar circuitos hamiltonianos en una gráfica sin pesos, ya que siempre se puede asignar a cada arista el peso 1. [9]Recordando el concepto de circuito Hamiltoniano: Este viene de uno de los problemas más viejos relacionados con la teoría de grafos. El problema de los puentes de Königsberg, que presenta la pregunta de si es posible dar un paseo en la ciudad de la figura 3.1 y regresar al punto de partida cruzando cada puente una vez.

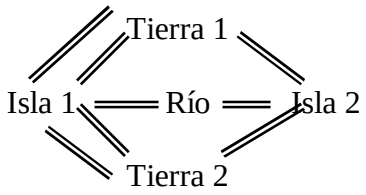


Figura 3.1 Grafo de Königsberg.

En 1936 el matemático suizo Leonhard Euler resolvió el problema. Construyó la gráfica de la figura 3.2 reemplazando la tierra firme por vértices y los puentes por aristas; es decir Tierra 1 es el vértice t_1, Tierra 2 es t_2, Isla 1 es I_1 y I_2 es Isla 2.

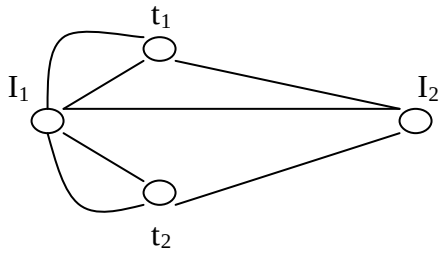


Figura 3.2

La pregunta fue entonces: ¿Existe un camino cerrado que contenga exactamente cada una de las aristas? Este camino se llama circuito de Euler o euleriano. Así un camino simple que contiene todas las aristas de una gráfica G se llama camino euleriano de G, o de otra manera se dice que un grafo conexo no vacío es un camino Eureliano si y sólo si no tiene vértices de grado impar. Y un camino euleriano cerrado se llama circuito euleriano.

[8] Garey Michael Computer and Intractability. 1979.
[9] Ross Kenneth Matemáticas Discretas 1990.

Recordando que la valencia de un vértice (x) en un grafo es el número de aristas (y) que confluyen en él val(x)=y, se da el siguiente teorema:

Teorema 1: En un grafo que tiene un circuito euleriano todos los vértices deben tener valencia par.

Corolario 1: Un grafo que tenga un camino euleriano tiene o bien 2 vértices de valencia impar, o bien no tiene vértices de valencia impar.

Un camino que contiene todos y cada uno de los vértices de G es llamado un camino Hamiltoniano de G, similarmente un ciclo Hamiltoniano de G es un ciclo que contiene cada vértice de G. Un grafo es Hamiltoniano si éste contiene un ciclo Hamiltoniano.

Dada la similitud en las definiciones de grafo Euleriano y grafo Hamiltoniano, y debido a que existe una caracterización particularmente útil de grafos Eulerianos, uno puede esperar un criterio análogo para los grafos Hamiltonianos, sin embargo no es el caso, dado que éste debe ser considerado como uno de los mayores problemas sin resolver de la teoría de grafos para desarrollar una caracterización aplicable de grafos hamiltonianos. Existen varias condiciones suficientes para establecer que un grafo es un grafo hamiltoniano.

Teorema 2: El grafo G de n vértices no tiene lazos ni aristas paralelas,
si $| V(G) | = n \geq 3$ y si $val(v) \geq n / 2$ para cada vértice v de G, entonces G es hamiltoniano.

Donde V(G) es el número de vértices del grafo G y val(v) la valencia del vértice v.

Teorema 3: Un grafo con n vértices y sin aristas paralelas ni lazos que tiene al menos un número de aristas igual a $(1/2)(n-1)(n-2) + 2$ es hamiltoniano.

Teorema 4: Suponer que el grafo G no tiene lazos ni aristas paralelas y que
$| V(G) | = n \geq 3$ Si $val(v) + val(w) \geq n$ para cada par de vértices v y w que no están conectados por una arista, entonces el grafo es hamiltoniano.

Algunos ejemplos de esto:

Ejemplo 1:

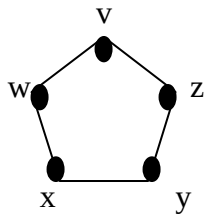
a) En la figura 3.3 a se muestra el grafo que tiene un circuito hamiltoniano v-w-x-y-z de cinco vértices.

b) En la figura 3.3.b se le añaden más aristas que no lo dañan, es decir, todo grafo completo K_n para $n \geq 3$ es hamiltoniano; además se puede ir de vértice a vértice en el orden que se quiera.

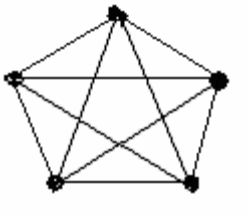
c) En la figura 3.3.c no se tiene circuito hamiltoniano ya que no hay ningún modo de recorrer todos los vértices.

d) En la figura 3.3.d se muestra que el grafo tampoco tiene camino Hamiltoniano, ya que del vértice superior a los dos inferiores se puede llegar pero no volver sin repetir vértices.

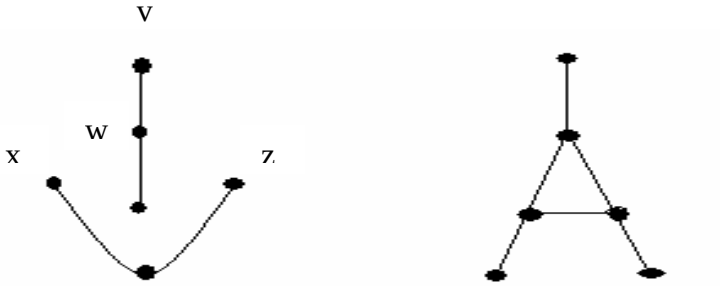
Un grafo H hamiltoniano con n vértices debe tener al menos n aristas. Esta condición es necesaria pero no suficiente, como lo ilustra la figura 3.3.d Desde luego, los lazos y las aristas paralelas no son utilizables.



(a)



(b)



(c) (d)

Figura 3.3 Circuitos Hamiltonianos.

Ejemplo 2:

a) El grafo K_5 de la figura 3.3 b tiene val(v) = 4 para cada v y tiene | V(G) | = 5 por lo que satisface la condición del teorema 2.

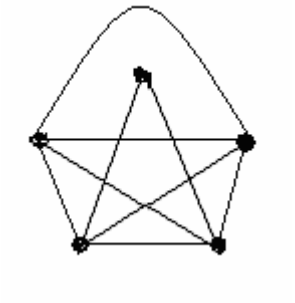
b) En la figura 3.4 cada uno de los grafos de la figura tiene | V(G) | / 2 = 5 / 2 y tiene un vértice de valencia 2. No satisfacen las hipótesis del teorema 2 y sin embargo son hamiltonianos.

El teorema 2 impone una condición uniforme a todos los vértices. El teorema 3 requiere solamente que existan suficientes aristas en alguna parte del grafo. Se establecen estas dos condiciones suficientes como consecuencia del teorema 4, el cual proporciona un criterio en términos de pares de valencias de vértices.

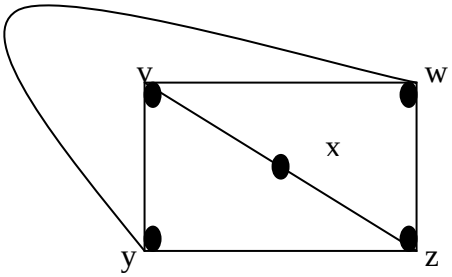
Ejemplo 3:

a) El grafo hamiltoniano de la figura 3.4 a tiene n = 5, lo que da (1/2) (n-1) (n-2) + 2 = 8. Tiene 9 aristas y por lo tanto satisface las hipótesis y también la conclusión del teorema 3.

b) El grafo hamiltoniano de la figura 3.4 b tiene también n = 5 y por lo tanto (1/2)(n-1)(n-2) + 2 = 8, pero tiene sólo 7 aristas. No satisface las hipótesis del teorema 3 ni las del teorema 2. Si no hubiera un vértice x en medio, se tendría K_4, con n = 4 por lo que (1/2)(n-1)(n-2) + 2 = 5, las 6 aristas que tendría serian más que suficientes. En estos términos el grafo satisface las hipótesis del teorema 4.



(a)



(b)

Figura 3.4 Circuitos hamiltonianos del ejemplo 3

Ejemplo 4: Para el grafo de la figura 3.4 b, n = 5. Hay tres pares de vértices distintos que no están conectados por ninguna arista. Se verifica que se cumplan las hipótesis del teorema 4 examinando:

Para < v, z>, val(v) + val(z) = 3 + 3 = 6 ≥ 5;
Para < w, x>, val(w) + val(x) = 3 + 2 = 5 ≥ 5;
Para < x, y>, val(x) + val(y) = 2 + 3 = 5 ≥ 5;

Es importante recalcar un tipo de grafo que será empleado para representar el problema del agente viajero en este trabajo, el grafo completo. Del que se dice: un grafo sin aristas paralelas ni lazos y en las cuales cada vértice está unido por una arista a cada uno de los otros vértices es llamado un grafo completo. Un grafo completo con n vértices tiene todos sus vértices con valencia n-1, por lo que es un grafo regular. Todos los grafos completos de n vértices son isomorfos unos a otros, y tienen un alto grado de simetría. Se muestra un ejemplo de un grafo completo n la siguiente figura.

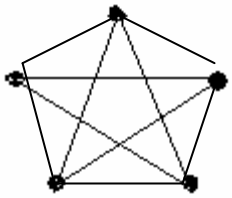


Figura: 3.5 Grafo Completo

Se tiene el siguiente grafo que representa a las ciudades 1, 2, 3, 4 con los pesos respectivos entre las ciudades:

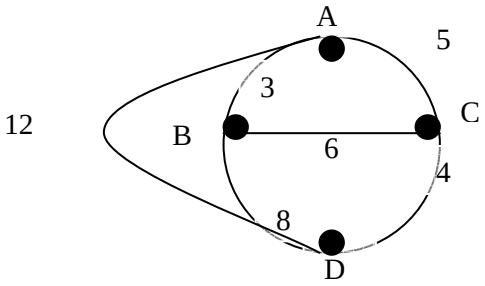


Figura 3.6 Grafo representativo de ruta

Se desea recorrer el grafo de manera que se toquen todos los vértices sin que se repitan, tomando como punto inicial al vértice A, se obtienen rutas y con ello un costo, finalmente se detecta el costo mínimo como se muestra a continuación:

Ruta	Nodos	Costo
1	A, B, C, D, A	3+6+4+12=25
2	A, B, D, C, A	3+8+4+5=20
3	A, D, B, C, A	12+8+6+5=31
4	A, D, C, B, A	12+4+6+3=25
5	A, C, B, D, A	5+6+8+12=31
6	A, C, D, B, A	5+4+8+3=20

Figura 3.7 Tabla de rutas

Como puede notarse el costo mínimo es 20, pero hay dos rutas que lo alcanzan, realizando un análisis se nota que es el mismo camino pero recorriendo las ciudades al revés.

La historia del problema del agente viajero desde el punto de vista algorítmico ha evolucionado gracias a que este problema ha sido una excelente plataforma de pruebas para la introducción de nuevas ideas algorítmicas como las relajaciones de Lagrange para el Recocido Simulado (Simulated Annealing), búsqueda Tabú, algoritmos Genéticos, Redes Neuronales, etc. Además se ha concentrado mucha investigación en el estudio de buenos algoritmos aunque no necesariamente óptimos en la solución. Se han probado problemas de la vida real tanto como problemas generados aleatoriamente con un número de ciudades que fluctúa entre 100 y 1000000 ciudades. La heurística *Spacefilling Curve* de Bartholdi y Platzman es muy rápida; para un problema del agente viajero geométrico de un millón de ciudades, este algoritmo llega a la solución con un 35 % de Held-Karp de la cota más baja en una ruta óptima en 22 segundos en una computadora SGI Challenge con 150 Mhz. El algoritmo Iterated Lin-Kernighan originalmente propuesto por Martín, Otto y Felten, para un millón de ciudades puede resolver el problema con una exactitud de 1.07 %, pero toma 500 horas en hacerlo, este mismo algoritmo para 1000 ciudades llega a la solución con una exactitud de 0.9 % en sólo 30 segundos. El algoritmo conocido como Branch and Cut ha sido un éxito en soluciones no triviales con 4416 ciudades.

Diferentes enfoques del problema del agente viajero

El problema del agente viajero puede ser visto desde distintos enfoques, de tal manera que si, el enfoque es:

A) Como un problema de optimización combinatoria:

Se tiene un grafo G en el cual cada una de las aristas tiene un peso positivo (o costo).

Solución Factible: Un ciclo que pase a través de cada vértice de G exactamente una vez; por ejemplo un circuito Hamiltoniano.

Función Objetivo: Para un circuito Hamiltoniano H de G se define

$$c(H)= \sum_{e \in H} c(e)$$

Solución Optima: Un circuito Hamiltoniano de costo mínimo.

B) Como un problema de decisión:

Un problema de decisión es aquel problema que pregunta si la respuesta es "si" o "no", así, un problema de decisión es especificado por describir una instancia típica del

problema y haciendo la pregunta de ¿cuál es ...? y ser contestada con un si o no. De manera que un algoritmo para resolver el problema del agente viajero como un problema de decisión sólo necesita contestar si o no, no es necesario dar una opción diferente o mejor; puede parecer restrictivo pero es una opción útil en muchos casos específicos.

Sea un grafo G con aristas como costos y un costo hipotético D.

Pregunta: ¿Es el recorrido G un circuito Hamiltoniano?

Solución: Decir si G es o no un circuito Hamiltoniano.

C) Como un problema euclidiano:

Se tiene un grafo G en el cual cada una de las aristas tienen un costo integral positivo, para el cual la desigualdad del triángulo se satisface para todos los vértices u, v, y w de tal modo que $c(u, v) \leq c(u, w) + c(w, v)$

Solución factible: Un circuito Hamiltoniano.

Función objetivo: Para un circuito Hamiltoniano H, se define un costo

$$c(H) = \sum_{e \in H} c(e)$$

Solución óptima: Un circuito Hamiltoniano de costo mínimo. [10]

Son muy importantes estas tres variaciones del problema del agente viajero normal, pero además existen otras variantes que amplían el enfoque del problema, se exponen algunas a continuación:

D) Extensión del Agente Viajero

Conocido con el nombre de Traveling Salesman Extensión (TSE). En este problema se tiene una ruta parcial, pero se desea saber si se puede generar el circuito completo sin exceder un cierto peso, por ejemplo se desea recorrer toda la República Mexicana, pero se tiene una parte de la ruta ya pensada, por lo que la pregunta sería si ¿se puede completar la ruta sin que cueste más de $5000?

Sea un conjunto finito $C = \{C_1, C_2, \ldots, C_m\}$ de ciudades, una distancia $d(C_i, C_j) \in Z^+$ para cada par de ciudades $C_i, C_j \in C$, una cota $B \in Z^+$, y una ruta parcial $\theta = < C_{\Pi(1)}, C_{\Pi(2)}, \ldots, C_{\Pi(k)} >$ de K distintas ciudades desde C, $1 \leq K \leq m$.

[10] Stinson D R An Introduction to the Design and Análisis of Algoritms 1987.

Pregunta: ¿Puede θ ser extendida para una ruta completa < $C_{\Pi(1)}$, $C_{\Pi(2)}$, ... , $C_{\Pi(k)}$, $C_{\Pi(k+1)}$, ..., $C_{\Pi(m)}$ > teniendo una longitud total B o menos?

E) Problema del Agente Viajero Simétrico.

Conocido como el Symmetric traveling salesman problem (TSP o PAV). Dado un conjunto de n nodos y distancias para cada par de nodos, encontrar una longitud total mínima que visite cada uno de los nodos exactamente una vez. La distancia del nodo i al nodo j es la misma que del nodo j al nodo i, en todos los casos.

F) Problema del Agente Viajero Asimétrico.

Conocido como Asymmetric Traveling Salesman Problem (ATSP). Dado un conjunto de nodos y distancias para cada par de nodos, encontrar una ruta de longitud total mínima que visite cada uno de los nodos exactamente una vez. En este caso, la distancia del nodo i al nodo j y la distancia del nodo j al nodo i podrían ser diferentes.

G) Problema del Agente Viajero Revisado.

Teorema: El problema de determinar una ruta óptima en el problema del Agente Viajero Simétrico con n ciudades es NP-Duro.

Pregunta: El problema del Agente Viajero simétrico puede ser visto como el problema de encontrar un ciclo hamiltoniano de costo mínimo, en un grafo no dirigido con pesos.

Demostración: Con esto en mente, es fácil probar que el problema de determinar la existencia de un ciclo hamiltoniano en un grafo no dirigido es transformable en el problema del agente viajero simétrico.

Dado un grafo no dirigido G=(V, E), |V| = n, se construye un problema del agente viajero asimétrico de n ciudades como sigue:

Sea V = {V_1, V_2, ..., V_n,}. El costo de ir entre las ciudades i y j es definido por

$$C_{ij} \begin{cases} 1 \text{ si } (v_i, v_j) \in E, \\ 2 \text{ de otra manera} \end{cases}$$

Claramente, el costo de una ruta óptima es n si G contiene un ciclo hamiltoniano; si G no contiene un ciclo hamiltoniano, el costo de una ruta óptima debe ser al menos n + 1 (n – 1 aristas de costo 1 y una arista de costo 2).

H) Problema de Ordenamiento secuencial.

Conocido con el nombre de Sequential Ordering Problem (SOP), este problema es un problema del Agente Viajero Asimétrico con limitaciones adicionales. Dado un

conjunto de n nodos y distancias para cada par de nodos, encontrar un camino Hamiltoniano del nodo 1 al nodo n de longitud mínima, camino que debe tomar en cuenta las limitaciones anteriores. Cada restricción precedente requiere de algún nodo i que tiene que ser visitado antes que algún otro nodo j.

I) Problema del Agente Viajero Geométrico.

Conocido como el Geometric Traveling Salesman Problem (GTS o PAVG). Se tienen ciudades en el plano y se quiere encontrar si hay un circuito solución del problema del agente viajero que no supere cierta distancia establecida. La distancia es euclidiana en términos de enteros (redondeo hacia arriba). Su definición matemática es:

Se tiene un conjunto $P \subseteq Z \times Z$ de puntos en el plano, y un entero positivo B.

Pregunta: ¿Existe un recorrido de longitud B o menor para el problema del agente viajero con C = P y $d((X_i, Y_j), (X_2, Y_2))$ igual a la distancia euclidiana discreta $[((X_i - X_2)^2 + (Y_1 - Y_2)^2)^{1/2}]$?

J) Cuello de botella del Agente Viajero.

Conocido como Bottleneck Traveling Salesman.

Sea un conjunto C de m ciudades, la distancia $d(C_i, C_j) \in Z^+$ para cada par de ciudades C_i , C_j elementos de C, y un entero positivo B.

Pregunta: ¿Existe una ruta de C cuya arista más larga no es mayor que B, por ejemplo una permutación $(C_1, C_2, \ldots, C_i)$ de C tal que $d(C_i, C_{i+1}) \leq B$ para $1 \leq i < m$ y tal que $d(C_m, C_1) \leq B$?

K) Problema del Agente Viajero cumpliendo la desigualdad del triángulo.

El problema del Agente Viajero es un problema que satisface la desigualdad del triángulo que consiste en $d(C_i, C_j) \leq d(C_i, C_k) + d(C_k, C_j)$ para toda tripleta de ciudades, dicho de otra manera, la distancia más corta entre dos ciudades en el plano es una línea recta, que corresponde a una ruta directa.

Es importante recalcar que, la desigualdad del triángulo se cumple en las versiones geométricas del problema del agente viajero (donde las ciudades corresponden a puntos en un espacio métrico y las distancias son calculadas de acuerdo a la métrica del espacio, ya sea esta euclidiana, rectilínea, u otra), y la distancia corresponde a la longitud del camino más corto entre dos puntos.

L) Problema del Cartero Chino.

Un cartero lleva el correo desde la oficina de correo hasta su destino, es decir lo reparte y después regresa a su oficina, él debe por supuesto, cubrir cada una de las calles en esa

área al menos una vez. Sujeto a esta condición, él desea escoger su ruta en tal forma que camine tan poco como sea posible. Este problema es conocido como el Problema del Cartero Chino, desde que fue considerado por el matemático chino Kuan en 1962. Representándolo en un grafo con peso se define el peso de una ruta $v_0e_1v_1$ ... e_nv_0 que será $\sum_{i=1}^{n} w(e_i)$.

Claramente el problema del cartero chino es justamente el encontrar un camino con peso mínimo en un grafo conexo con pesos que no deben ser negativos. Refiriéndose a este camino como el camino óptimo. Si G es Euleriano, entonces cualquier camino Euleriano de G es un camino óptimo porque el camino Euleriano es un camino que navega a través de cada vértice exactamente una vez. El problema del cartero chino se resuelve fácilmente en este caso ya que existe un buen algoritmo para determinar un camino Euleriano en un grafo Eureliano. El algoritmo se debe a Fleury, donde construye un camino Euleriano trazando un rastro, sujeto a una condición, que en cualquier estado, una arista de un subgrafo no-trazado es tomado solamente si no hay alternativa.

M) Cartero Rural.

Conocido como Rural Postman.

Instancia: El grafo G = (V, E), de longitud l (e) $\in Z_0^+$ para cada e $\in$ E, subconjunto E' $\subseteq$ E, limite B $\in Z^+$.

Pregunta: ¿Existe un circuito en G que incluye cada arista en E' y que tiene una longitud total no mayor que B?

Para la siguiente clasificación se exponen primero algunos conceptos de la teoría de gráficas.

Camino: Es un recorrido por las aristas de un gráfico, los caminos permiten la repetición de vértices.

Camino cerrado: Es un camino que inicia y termina en el mismo vértice.

Circuito: Es un camino cerrado que no repite ningún vértice.

Circuito dirigido: Es un circuito en un dígrafo o grafo dirigido, que es el que tiene una dirección o sentido en cada arista.

Así un camino que contiene todos y cada uno de los vértices de un grafo G es llamado un *camino Hamiltoniano de G*, similarmente un *Circuito Hamiltoniano de G* es un circuito que contiene cada vértice de G. Un grafo es Hamiltoniano si contiene un circuito Hamiltoniano.

Con esto en mente se estudian las últimas divisiones del problema del agente viajero.

N) Problema del circuito Hamiltoniano.

También conocido como Hamiltonian cycle problem (HCP) y que consiste en:

Dado un grafo, probar que el grafo contiene o no un ciclo Hamiltoniano.

Instancia: Un grafo G = (V, E)

Pregunta: ¿G contiene un circuito hamiltoniano?

Teorema: Un circuito hamiltoniano es NP-Completo.

1) Problema del camino Dirigido Hamiltoniano.

Conocido como Directed Hamiltonian Circuit.

Sea un grafo dirigido G = (V, A)

Pregunta: ¿G contiene un circuito dirigido hamiltoniano?

2) Problema del camino Hamiltoniano.

Conocido como Hamiltonian Path.

Sea un grafo G = (V, E)

Pregunta: ¿G contiene un camino hamiltoniano?

3.2 Aplicaciones.

Obviamente las aplicaciones del problema del Agente Viajero surgen en el direccionamiento de servicios de rutas de entrega o visitas, pero en muchos de estos casos no está prohibido regresar a un nodo ya visitado con anterioridad. Se han encontrado aplicaciones en contextos mucho menos obvios, el más común implica el orden en que se deben procesar productos diferentes en una instalación de manufactura, como pudiera ser una línea de ensamblado.

Por ejemplo, un fabricante de fregaderos para cocina produce cerca de veinte modelos diferentes en una línea de ensamblado continua. Algunos de los modelos son muy semejantes a otros; pero hay otros que son bastante diferentes. El costo de un “cambio” en la línea de ensamblado de un modelo a otro depende de las características de estos modelos.

En algunos casos, los cambios (arranques) se pueden hacer muy rápido a un costo bajo; en otras, se requiere una buena cantidad de tiempo y mano de obra. Además puede tardar más un cambio del modelo A al B que a la inversa. Un cambio en una dirección puede eliminar algunas operaciones; el otro puede implicar algunas adicionales, tales como un incremento en el número de entrepaños en el gabinete debajo del fregadero.

Determinar el orden para producir los modelos de tal manera que se minimicen los costos de arranque es un problema asimétrico del agente viajero. Los costos de los cambios entre los modelos son análogos a las distancias entre los puntos. Cada modelo debe producirse una sola vez y la producción debe regresar al primer modelo.

Este tipo de problema de arranque mínimo surge en una gran variedad de contextos, dondequiera que un trabajo se ejecute en una sola instalación y siempre que se involucren arranques diferentes.

3.3 El procedimiento de Little.

Este procedimiento iterativo fue desarrollado por Little, Murry, Sweeney y Karel en 1963. Generalmente se considera como eficiente, aunque el tiempo de cálculo (a mano o con computadora) crece rápidamente al aumentar el tamaño del problema.
Este procedimiento se conoce también como bifurcar y limitar o ramificar y acotar (branch and bound).

El método está basado en un árbol de búsquedas, donde en cada etapa todas las posibles soluciones del problema son divididas en dos subconjuntos, cada uno representado por nodos en un árbol de decisiones. Al dividir en dos subconjuntos en cada etapa, el subconjunto de la izquierda contendrá una arista especifica (i,j) y el subconjunto de la derecha no tendrá esa arista. Después de ramificar se calculan las cotas para cada uno de los subconjuntos; en el próximo espacio de soluciones en que se busque se debe escoger sólo uno de ellos, que es el mínimo de las dos cotas a ser comparadas. Este proceso se repite recursivamente hasta encontrar el ciclo hamiltoniano. Entonces, solamente dentro de los subconjuntos de soluciones se buscará la cota que sea más baja que el valor de la cota inicial o de arranque.

Esta forma de ramificar y acotar la solución nos permite descartar un buen número de subconjuntos de solución, esto evita realizar una búsqueda infructuosa. La heurística consiste en reducir en el proceso computacional las cotas más bajas, dicha reducción da origen al método Una mejor Ramificación y Acotamiento (A Better Branch and Bound). Para que sea aplicable, el costo debe estar bien definido, es decir la matriz de adyacencia debe representar un grafo completo.

Primero se describe éste proceso de manera abstracta y posteriormente se ilustra mediante un ejemplo numérico para a continuación enunciar el algoritmo con sus procesos en pseudocódigo.

Los pasos del algoritmo son:

1.- Redúzcase la matriz de costos hasta que haya un cero en cada fila y en cada columna. Restando el elemento más pequeño de cada fila y cada columna respectivamente.

2.- Para cada elemento cero en la matriz se registra una multa por no utilización, dicha multa será igual a la suma de los elementos más pequeños de renglón y columna.

3.- Se elige el elemento con la multa mayor, rompiendo arbitrariamente los empates y se divide el conjunto de las posibles soluciones en dos subconjuntos, los que contiene el enlace de dicho elemento y los que no.

4. Se calculan los límites inferiores de los costos de todas las rutas de cada subconjunto

5.- Selecciónese uno de los subconjuntos para particiones adicionales, volviendo al paso dos para volver a conseguir que exista un elemento cero en cada fila y columna.

6. La casilla que tenga la multa mayor será utilizada nuevamente para subdividir el conjunto de soluciones.

7.- Se vuelven a calcular los límites inferiores de cada subconjunto, y se continúan repitiendo los pasos desde el dos. Naturalmente el algoritmo termina cuando la matriz de costos no se puede dividir más y se ha obtenido la trayectoria que recorre todas las ciudades.

Ahora se verá este procedimiento ilustrado en el siguiente ejemplo:

Consideremos el problema del agente viajero para seis ciudades, con el costo del viaje entre cada una de ellas dado por la siguiente matriz de adyacencia, donde Vi son los vértices, nodos o ciudades del problema y la matriz debe contener infinitos (α) en la diagonal principal debido a que no se permite ir de una ciudad a ella misma en un solo paso.

i,j	V1	V2	V3	V4	V5	V6
V1	α	3	93	13	33	9
V2	4	α	77	42	21	16
V3	45	17	α	36	16	28
V4	39	90	80	α	56	7
V5	28	46	88	33	α	25
V6	3	88	18	46	92	α

Figura 3.8 Matriz de adyacencia de 6 ciudades para el PAV.

Se analiza ahora el proceso de reducción, un ciclo hamiltoniano de longitud n contiene exactamente un elemento de cada fila de la matriz de costo o adyacencia W y exactamente un elemento de cada columna de W. Si a la matriz W(i,j) se le resta una constante q de cualquier fila o de cualquier columna, el costo de todos los ciclos Hamiltonianos (circuitos) son reducidos por q. Más aún los costos relativos de los diferentes ciclos restantes también son reducidos por q, lo mismo que el circuito óptimo. Si tal sustracción es hecha de las filas y las columnas, de tal manera que cada fila y cada columna contengan al menos un cero pero además que no guarde elementos negativos, entonces la cantidad restada será la cota más baja en el costo de cualquier solución, en este proceso reside la reducción que mejora el algoritmo.

Se comienza ahora a resolver el PAV, primero se reduce la matriz hasta que haya un cero en cada fila y cada columna. Esto se consigue restando el elemento más pequeño de cada fila de cada uno de los elementos de ella, luego restando el elemento más pequeño de cada columna, de la matriz restante, de cada uno de los elementos de dicha columna.

Observe las siguientes tablas que ilustran los pasos descritos anteriormente:

i,j	V1	V2	V3	V4	V5	V6	Menor
V1	α	3	93	13	33	9	3
V2	4	α	77	42	21	16	4
V3	45	17	α	36	16	28	16
V4	39	90	80	α	56	7	7
V5	28	46	88	33	α	25	25
V6	3	88	18	46	92	α	3

a) Matriz de adyacencia y menores de cada fila.

I,j	V1	V2	V3	V4	V5	V6	Menor
V1	α	0	90	10	30	6	3
V2	0	α	73	38	17	12	4
V3	29	1	α	20	0	12	16
V4	32	83	73	α	49	0	7
V5	3	21	63	8	α	0	25
V6	0	85	15	43	89	α	3
							58

b) Matriz de adyacencia después de haber restado los menores de cada fila.
Suma de menores: 3 + 4 + 16 + 7 + 25 + 3 = 58

I,j	V1	V2	V3	V4	V5	V6	Menor
V1	α	0	90	10	30	6	3
V2	0	α	73	38	17	12	4
V3	29	1	α	20	0	12	16
V4	32	83	73	α	49	0	7
V5	3	21	63	8	α	0	25
V6	0	85	15	43	86	α	3
Menor	0	0	15	8	0	0	81

c) Matriz de adyacencia con menores de fila y columna.
Suma de menores de fila y columna = 81.

i,j	V1	V2	V3	V4	V5	V6
V1	α	0	75	2	30	6
V2	0	α	58	30	17	12
V3	29	1	α	12	0	12
V4	32	83	58	α	49	0
V5	3	21	48	0	α	0
V6	0	85	0	35	89	α

d) Matriz de adyacencia después de restar menores de cada columna.
Cota actual: 81

Figura 3.9

Ahora la pregunta sería ¿Cómo se divide el conjunto de todas las soluciones en dos clases? Para realizar esto se registra una multa P_{hk} para cada elemento cero en la matriz W(i, j). Se considera que si no se utiliza el segmento (h, k) se debe usar algún elemento de la fila h y alguno de la columna k, excluyendo h, k, por lo tanto,

$$P_{hk} = \text{mín } W_{h,\,j} + W_{i,\,k}$$

Se anota el resultado de cada operación en la esquina superior izquierda de cada casilla que tenga cero. Por ejemplo considérese el cero en (2, 1), la suma del mínimo de la fila 2 y la columna 1, excluyendo el cero en (2, 1) es 12 + 0 =12. Para (6, 1) la suma es 0 + 0 = 0. Del mismo modo se realizan los cálculos para los demás ceros.

(1, 2) = 2 + 1 = 3 (2, 1) = 12 + 0 = 12 (3, 5) = 1 + 17 = 18
(4, 4) = 32 + 0 = 32 (5, 6) = 0 + 2 = 2 (6, 1) = 0 + 0 = 0
(6, 3) = 0 + 0 = 0 (6, 3) = 0 + 48 = 48

i,j	V1	V2	V3	V4	V5	V6
V1	α	0 0	75	2	30	6
V2	0 0	α	58	30	17	12
V3	29	1	α	12	18 0	12
V4	32	83	58	α	49	32 0
V5	3	21	48	2 0	α	0 0
V6	0 0	85	48 0	35	89	α

Figura 3.10 Matriz con las multas por no utilización.

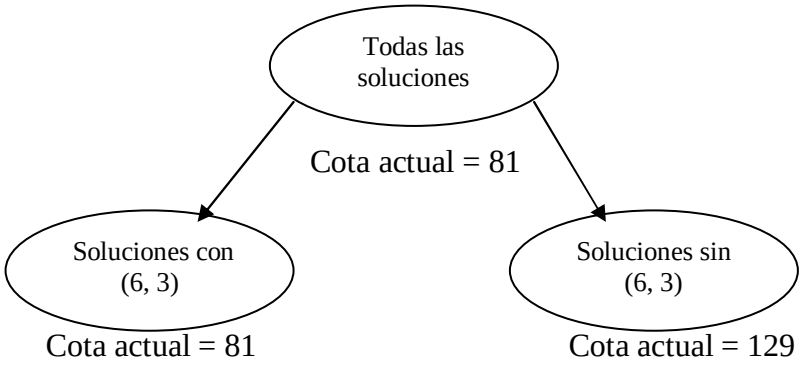
A continuación se elige el elemento cero con la mayor penalización, en este caso (6, 3) si se presentara un empate se realiza la selección arbitrariamente. Ahora se divide el espacio de soluciones en dos, generando dos matrices.

La matriz derecha será el conjunto que contendrá todas las soluciones que excluyen a los nodos (6,3), por lo tanto sabiendo que (6, 3) son excluidos se debe cambiar el valor de la matriz en (6, 3) a ∞, lo que implica restarle al renglón 6 y la columna 3 los menores valores excluyendo el de (6, 3), es decir que se le resta cero al renglón 6 y 48 a la columna 3, obteniendo la cota actual más baja de 81 + 48 = 129 para todas las soluciones que excluyen a (6, 3).

La matriz izquierda mientras tanto contiene todas las soluciones que incluyen los nodos (6, 3), por lo que la sexta fila y la tercer columna deben ser borradas de la matriz de costos reducida porque ahora ya nunca se podrá ir desde el nodo seis a ningún otro nodo o llegar al nodo tres desde ningún otro nodo.

El resultado es una matriz de costos de dimensión (5X5) que es menor en uno que la anterior de (6X6), y más aun ya que todas las soluciones de este subconjunto usan a los nodos (6, 3) los nodos (3, 6) no serán usados nunca más por lo que se debe cambiar el valor de la matriz en (3, 6) a ∞ para prohibir este nodo.

El árbol de búsqueda binario en este momento queda como se presenta en la siguiente figura con su respectiva matriz derecha e izquierda.



Solución con (6, 3)

i, j	V1	V2	V4	V5	V6
V1	∝	0	2	30	6
V2	0	∝	30	17	12
V3	29	1	12	0	∝
V4	32	83	∝	49	0
V5	3	21	0	∝	0

Solución sin (6, 3)

i, j	V1	V2	V3	V4	V5	V6
V1	∝	0	27	2	30	6
V2	0	∝	10	30	17	12
V3	29	1	∝	12	0	12
V4	32	83	10	∝	49	0
V5	3	21	0	0	∝	0
V6	0	85	∝	35	89	∝

Figura 3.11

Los nodos (6, 3) fueron usados para la solución porque, de todos los nodos, éstos causaron el mayor incremento en la cota actual más baja del subárbol derecho.

De manera que la solución óptima se encontrara en las ramas izquierdas más que en las ramas derechas, las ramas izquierdas reducen la dimensión del problema, sin embargo las ramas derechas sólo agregan otro ∝ y quizás otros pocos ceros sin cambiar la dimensión de la matriz.

Así se vuelven a calcular las multas por no utilización para la matriz de (5X5) que representa el problema del subárbol izquierdo, como lo indica la tabla de la siguiente página.

i, j	V1	V2	V4	V5	V6
V1	∝	3 0	2	30	6
V2	15 0	∝	30	17	12
V3	29	1	12	18 0	∝
V4	32	83	∝	49	32 0
V5	3	21	0 0	∝	0 0

Figura 3.12 Matriz con multas por no utilización.

(1, 2) = 2 + 1 = 3

(2, 1) = 12 + 3 = 15

(3, 5) = 1 + 17 = 18

(4, 6) = 32 + 0 = 32

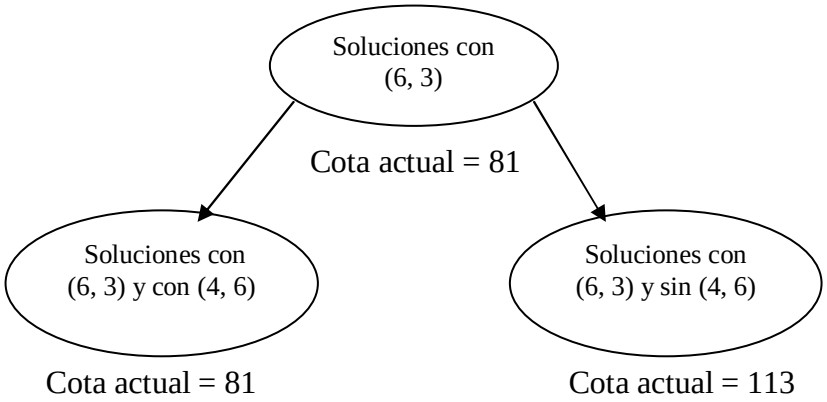
(5, 4) = 0 + 2 = 2

(5, 6) = 0 + 0 = 0

Por lo tanto la siguiente división se hará con los nodos (4, 6). Esto incrementa la cota más baja de todas las soluciones que incluyen al nodo (6, 3) y que excluyen los nodos (4, 6) a 81 + 32 = 113.

Esto decremento la dimensión de la matriz en la rama izquierda a una matriz de (4 X 4), ya que debe borrarse la fila 4 y la columna 6.

Esta situación se presenta en la imagen siguiente donde se muestra el árbol generado hasta el momento con su respectiva matriz derecha e izquierda.



Solución con (6, 3) y (4, 6)

i, j	V1	V2	V4	V5
V1	∝	0	2	30
V2	0	∝	30	17
V3	29	1	∝	0
V5	3	21	0	∝

Solución con (6, 3) y sin (4, 6)

i, j	V1	V2	V4	V5	V6
V1	∝	0	2	30	6
V2	0	∝	30	17	12
V3	29	1	12	0	∝
V4	0	51	∝	17	∝
V5	3	21	0	∝	0

Figura 3.13

Nótese que ya que los nodos (4, 6) y (6, 3) son incluidos en la solución, los nodos (3, 4) no se usaran nunca más, esto se refuerza poniendo como valor en (3, 4) = ∝. En general si el nodo que se agrega a la ruta parcial va desde i_n a j_l y la ruta parcial contiene caminos (i_1, i_2, ..., i_n) y (j_1, j_2, ..., j_k), los nodos cuyo uso deben prohibirse están en (j_k, i_l).

Se realiza el nuevo cálculo de multas con los resultados siguientes:

I, j	V1	V2	V4	V5
V1	∝	3 0	2	30
V2	20 0	∝	30	17
V3	29	1	∝	18 0
V5	3	21	5 0	∝

Figura 3.14

(1, 2) = 2 + 1 = 3
(2, 1) = 17 + 3 = 20
(3, 5) = 1 + 17 = 18
(5, 4) = 3 + 2 = 5

Concluyendo que los mejores nodos son (2, 1), así que a la matriz de costos derecha se le prohíben los nodos (2, 1) al hacer su valor = ∝ y además restarle 17 a la segunda fila y 3 a la primera columna. Mientras que en la matriz izquierda se eliminan el renglón 2 y la columna 1 y colocando el valor ∝ en la casilla (1, 2). Se muestran solamente las dos matrices resultantes.

Solución con (6, 3), (4, 6) y (2, 1)

i, j	V2	V4	V5
V1	∝	2	30
V3	1	∝	0
V5	21	0	∝

Solución con (6, 3), con (4, 6) y sin (2, 1)

i, j	V1	V2	V4	V5
V1	∝	0	2	30
V2	∝	∝	13	0
V3	26	1	∝	0
V5	0	21	0	∝

Figura 3.15

Después de dividir el espacio de búsqueda en la matriz derecha e izquierda, la matriz de costos izquierda es una matriz de (3 X 3), se realiza nuevamente la reducción del circuito hamiltoniano ya que siempre debe haber por lo menos un cero en cada renglón y columna, además de no existir elementos negativos. Se calculan los menores de renglón, se restan y calculan los menores de columna para a continuación restarlos, el proceso se ilustra a continuación.

	V2	V4	V5	Menores
V1	∝	2	30	2
V3	1	∝	0	0
V5	21	0	∝	0

Figura 3.16

Matriz con menores de renglón. Suma de menores 2.

	V2	V4	V5
V1	∝	0	28
V3	1	∝	0
V5	21	0	∝

Matriz después de restar los menores de renglón.

	V2	V4	V5	Menores
V1	∝	0	28	2
V3	1	∝	0	0
V5	21	0	∝	0
Menores	1	0	0	3

Figura 3.17

Matriz con menores de renglón y columna. Suma de menores 3.

	V2	V4	V5
V1	∝	0	28
V3	0	∝	0
V5	20	0	∝

Figura 3.18

Matriz después de restar los menores de columna. Cota Actual 81 + 3 = 84

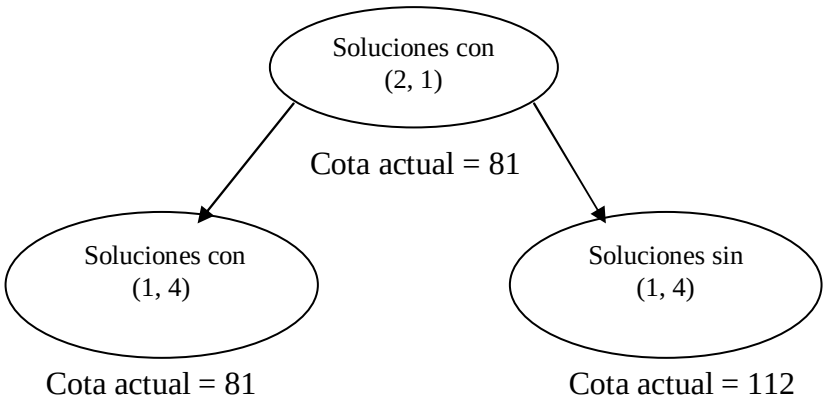
Se calculan nuevamente las multas para los elementos cero.

	V2	V4	V5
V1	∝	28 / 0	28
V3	20 / 0	∝	28 / 0
V5	20	20 / 0	∝

Figura 3.19

Se presenta un empate, eligiendo los nodos (1, 4) arbitrariamente.

Obteniendo:



Solución con (1, 4)
Cota actual = 84

	V2	V5
V3	0	0
V5	20	∝

Solución sin (1, 4)
Cota actual = 112

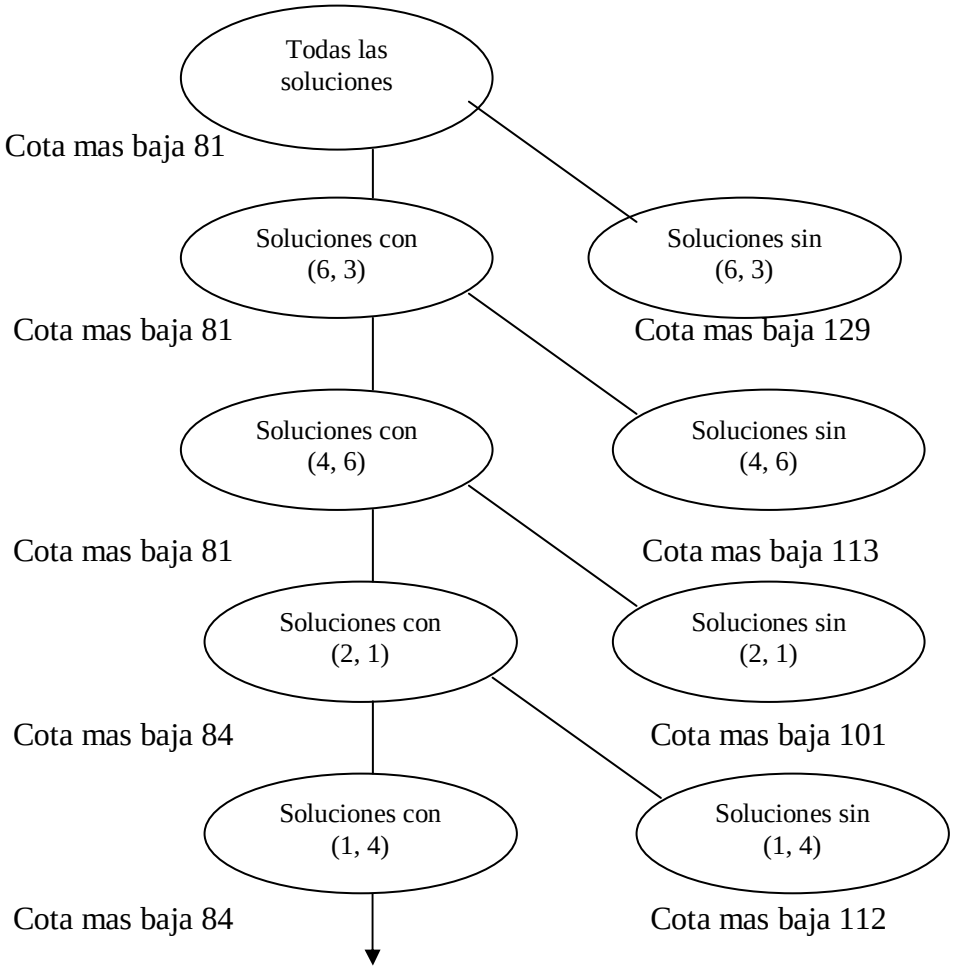
	V2	V4	V5
V1	∝	∝	0
V3	0	∝	0
V5	20	0	∝

Figura 3.20

En este punto se nota que la matriz de costos es de 2 X 2, en este momento las dos últimas parejas de nodos se pueden forzar para obtener un camino. En este ejercicio se tienen los nodos (6, 3), (4, 6), (2, 1) y (1, 4) así que sólo falta agregar los nodos (3, 5) y (5, 2) para completar la ruta que resuelva el problema del agente viajero. Con la historia de los nodos eliminados se encuentra la ruta final siguiendo las direcciones. En este caso es 1, 4, 6, 3, 5, 2, 1 cuyo costo es de 104. Los nodos de búsqueda se muestran a continuación. Al analizar el costo y las cotas más bajas del árbol binario puede notarse que existe un subárbol con la cota actual más baja (101) que es menor que el costo de la ruta que se obtuvo como solución por lo que ese subárbol debe ser examinado y expandido, ya que existe la posibilidad de que haya otra ruta con un costo menor que el de la ruta anterior. La cota actual más baja 101 incluye los nodos (6, 3) y (4, 6) pero excluye los nodos (2, 1), la matriz de costos se presenta a continuación.

i, j	V1	V2	V4	V5
V1	∝	0	2	30
V2	∝	∝	13	0
V3	26	1	∝	0
V5	0	21	0	∝

Figura 3.21



Solución 1, 4, 6, 3, 5, 2, 1

Costo 104

Figura 3.22

Árbol binario de búsqueda del procedimiento de Little.

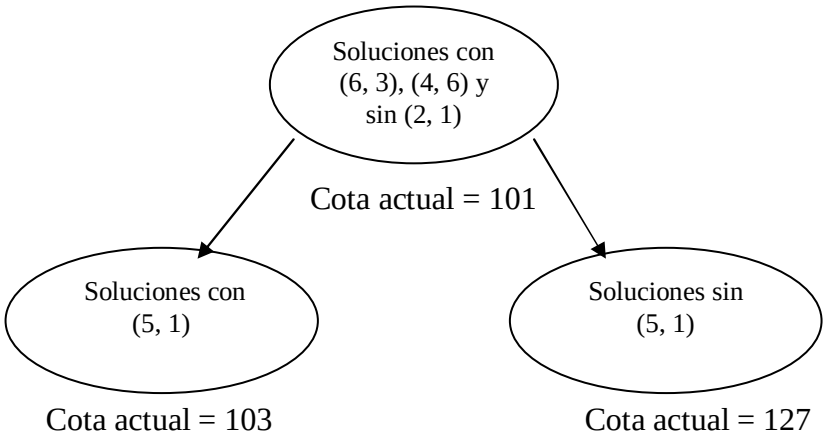
Ahora se calculan las multas de la matriz de la cota 101.

i, j	V1	V2	V4	V5
V1	∞	31	2	1
V3	26	∞	13	30

V5	26	0	21	2	0	∝

Figura 3.23

El nodo a eliminar es (5, 1) esto excluye dicho nodo y suma 21 a la cota actual dando 101 + 26 = 127, generando el subárbol siguiente:



Solución con (1, 4) y sin (2, 1) con (5, 1)

	V2	V4	V5
V1	0	2	∝
V2	∝	13	0
V3	1	∝	0

Solución con (6, 3) con (4, 6), sin (2, 1) y sin (2, 1)

	V1	V2	V4	V5
V1	∝	0	2	30
V2	∝	∝	13	0
V3	0	1	∝	0
V5	∝	21	0	∝

Figura 3.24

Solución 1, 4, 6, 3, 2, 5, 1
Costo 104

De esta manera se obtienen dos soluciones óptimas y dos rutas diferentas (1, 4, 6, 3, 5, 2, 1) y (1, 4, 6, 3, 2, 5, 1) cada una con un costo de 104 que es el costo menor de todas las rutas. Como puede verse no necesariamente se obtiene una sola ruta como solución óptima, puede haber mas de una solución óptima con un solo costo mínimo y rutas diferentes. También cabe aclarar que no siempre la solución óptima se encuentra en la primer solución encontrada como ocurrió en el ejemplo, en este caso especial se examinaron 13 nodos lo que es un gran ahorro de búsqueda.

Se mencionó antes, que la matriz después de reducir su dimensión una y otra vez llegaba a una matriz de 2 X 2. Entonces se forzaban los últimos pares de nodos para cerrar la ruta correspondiente dado que ninguna otra división del espacio de búsqueda podía realizarse. Así que cuando esto ocurre simplemente se agregan los últimos nodos sin que se puedan seleccionar y se forma el ciclo hamiltoniano.

Obsérvese ahora que este algoritmo puede dividirse en varios subalgortimos que resuelven tres preguntas importantes, que determinan la heurística a seguir:

A) ¿Cómo acotar los pesos de la solución de cada subárbol?

B) ¿Cómo se representa el conjunto de soluciones?

C) ¿Cómo se divide y cómo se escogen los mejores nodos en el espacio de búsqueda?

Se responde a estas preguntas conforme se describen las etapas a seguir.

Primero se acotan las soluciones, es decir se reduce la matriz W restando una constante de manera que los valores remanentes no sean negativos. Esto cambiará los pesos de cada viaje, pero no el conjunto de viajes legales y sus pesos relativos.

A continuación la función *Reduce* que responde a el inciso a.

```
1.Función Reduce (w)
2. Entero sumamenores // Guarda la suma de los menores de las columnas y renglones
3. ntamañoR // número de renglones de W
4. ntamañoC // número de columnas de W
5. filred() // Función que obtiene el menor elemento de la fila especificada
6. colred() // Función que obtiene el menor elemento de la columna especificada
7. restaR(,) // función que resta el segundo parámetro a cada elemento del renglón
8. // especificado en el primer parámetro.
9. restaC(,) // función que resta el segundo parámetro a cada elemento de la columna
10. // especificada en el primer parámetro.
```

```
11.begin
12.sumamenores=0
13. // Calcula y resta el menor de cada renglón
14. for i = 1 to ntamañoR
15.         begin
16.                 filred (i) // menor de los elementos de la fila i
17.                 if filred(i) > 0 then
18.                         begin
19.                                 restaR (i, filred(i)) //de cada elemento de la fila i
20.                                 sumamenores = sumamenores + filred(i)
21.                         end
22.                 end If
24.         end
24. next I
25. // Calcula y resta el menor de cada renglón
26. for j = 1 to ntamañoC
27.         begin
28.                 colred (j) = menor de los elementos de la columna j
29.                         if colred(j) > 0 then
30.                                 begin
31.                                         resta colred(j) de cada elemento de la fila j
32.                                         sumamenores = sumamenores + colred(j)
34.                                 end
34.                         end If
35.         end
36. next j
37.regresa sumamenores
38. End
39.// Fin de la función Reduce
```

A continuación describen las partes más importantes:

En la línea 2 está *sumamenores* que guardará el valor de la reducción.

En las líneas 3 y 4 están *ntamañoR* y *ntamañoC* que es el tamaño de los renglones y columnas de la matriz de adyacencia w.

De la línea 14 a 24 se localiza el menor de cada fila i guardándolo en *filred*(i), y se resta este menor de cada elemento de la matriz W en la fila i, tantas veces como filas existan. Acumulando en *sumamenores* los menores de todas las filas.

De la línea 26 a la 36 se realiza la misma operación con las columnas guardando los menores de cada columna en *colred*(j), acumulando en *sumamenores* los menores elementos de todas las columnas.

La matriz de adyacencia o de costos es usada para representar el conjunto de soluciones, lo que responde el inciso b. Para ramificar o dividir el espacio de soluciones se debe determinar el mejor cero, es decir en que posición (r, c) se encuentra el cero que es capaz de reducir la matriz w cuando se coloca un $\propto$ de manera que maximice la cantidad a ser restada desde la fila y la columna. La posición del mejor cero determina cuales son los mejores nodos. Esto responde la mitad del inciso c y se lleva a cabo con el siguiente procedimiento:

```
1.  Procedimiento selecciona_los_mejores_nodos ()
2.  A // matriz de adyacencia
3.  ntamañoR // número de renglones
4.  ntamañoC //
5.  cotamejor
6.  total // suma del menor de fila y el menor de columna
7. begin
8. cotamejornd = -∝
9. for i = ntamaño    // Fila
10.     for j = ntamaño    // Columna
11.          if A(i, j) = 0 then
12.              begin
13.              buscar menor_en_fil
14.              buscar menor_en_col
15.            total = menor_en_fil + menor_en_col
16.            if total > cotamejornd then
17.                begin
18.                cotamejornd = menor_en_fil + menor_en_col
19.                r = i
20.                c = j
21.                end
22.            endif
23.            end
24.       endif
25.       A(i, j) – menor_en_fil
26.     Next j
27.     A(i, j) – menor_en_col
28. Next i
29. End del procedimiento selecciona_mejor_cero
```

De la línea 9 a la 15 se localiza el mejor cero. Se localiza un cero en A (i, j) en la fila i, en *menor_en_fil* se guarda el menor de la fila exceptuando A(i, j) que tiene 0. Es decir se localiza el menor de cada fila diferente de cero. Si hay dos o más ceros el menor índice se toma como posición del cero. En la columna j donde se localizó el cero, en *menor_en_col* se guardan el menor de la columna exceptuando A(i, j) que tiene 0. Es decir se localiza el menor de cada columna diferente de cero. Si hay dos o más ceros el menor índice se toma como posición del cero.

En las líneas 16 a 24 Ocurre que si la el total, el cual es la suma de *menor_en_fil* y *menor_en_col* resulta ser mayor que la *cotamejornd*, se toma esta suma como la nueva *cotamejornd*. Guardando además en r el índice de la fila que será uno de los mejores nodos y en c al índice de la columna que será el otro mejor nodo.

La línea 25 resta a la matriz de costos reducida A(i, j) *menor_en_fil* de cada elemento de la fila r, excepto a ceros e infinitos.

La línea 27 resta a la matriz de costos reducida A(i, j) *menor_en_col* de cada elemento de la columna c, excepto en ceros e infinitos.

Tanto la función *Reduce* como el procedimiento *selecciona_mejor_cero* son llamados en el procedimiento recursivo llamado *Acotar* que contesta la otra mitad del inciso c, como se describe a continuación.

```
1.  Procedimiento Acotar (numnds, w, new_reg)
2.  w // Matriz de adyacencia
3.  cotaactual // Número que representa la actual cota para la división de las soluciones
4.  cotainf // Número que representa la cota inferior
5.  n_vértices // Número de vértices del problema
6.  numnds // Número de renglones
7.  new_reg // Nuevo renglón
8.  NewA // Nueva matriz después de borrar los renglones y columnas
9. begin
10. cotaactual = cotaactual + reduce (A)
11. if cotaactual < cotainf then
12.     if numnds = n_vértices – 2 then
13.      begin
14.      // Los dos últimos nodos son forzados
15.      // Se guarda la nueva solución
16.      cotainf = cotaactual
17.    end
18.   else
19.     begin
20.      seleccionar_mejor_cero (ntamaño,r,c,a,cotamejor)
21.      // Se prevén y prohíben subciclos
22.      NewA = A – columna c – fila r
23.      Acotar (numnds + 1, New_A, new_reg)
24.      restaura A agregando columna c y fila r
25.      if cotamasbaja < cotainf then
26.        begin
27.        A (r, c) = INFINITO
28.        Acotar (numnds, a, new_reg)
29.        a (r, c) = 0
30.        end
31.      endif
```

32. end
33. restaura_matriz_izq de la reducción
34. end del procedimiento Acotar

A continuación la descripción de las líneas más importantes:

Línea 22 en NewA se almacena la matriz A después de borrar la fila r y la columna c.

Línea 23 se genera el subárbol izquierdo al hacer el llamado recursivo al procedimiento Acotar con el número de nodos más uno y la matriz New_A.

Línea 28 aquí la segunda recursividad manda a la matriz derecha "a" con el mismo número de nodos.

Línea 29 se restaura en la matriz "a" los nodos excluidos que fueron en su oportunidad la dirección del mejor cero haciendo a la matriz en la fila r y la columna c igual a cero.

El método de Little es en el peor de los casos básicamente un método de búsqueda exhaustiva, en ese caso se examinaría todo el árbol para analizar todas las posibles soluciones.

Para un problema del agente viajero asimétrico de n ciudades hay (n – 1)! Ciclos Hamiltonianos diferentes; en un caso típico sin embargo, la situación no es tan mala.

Por ejemplo para el caso especial de seis ciudades estudiado anteriormente se examinaron solo 13 nodos de los 120 ciclos distintos para este problema de 6 ciudades [(n – 1!) = (6 – 1)! = 120]. Por lo que fueron 13 reducciones, 13 divisiones del espacio de búsqueda, etc, que es mucho menor que 120 = 5!

De hecho el tiempo de ejecución es extremadamente dependiente de la instancia del problema del agente viajero.

3.4 Trayectoria mínima de redes.

Otro enfoque del problema del agente viajero es aquél en que se desea ir de un lugar a otro o a varios otros, y al llegar a cada uno de ellos se debe seleccionar entre varias trayectorias que involucran diferentes lugares de parada a lo largo del camino.

Por ejemplo, supóngase que se desea ir de A hacia K en la siguiente red, donde los todos los enlaces tienen doble sentido a menos que se especifique lo contrario.

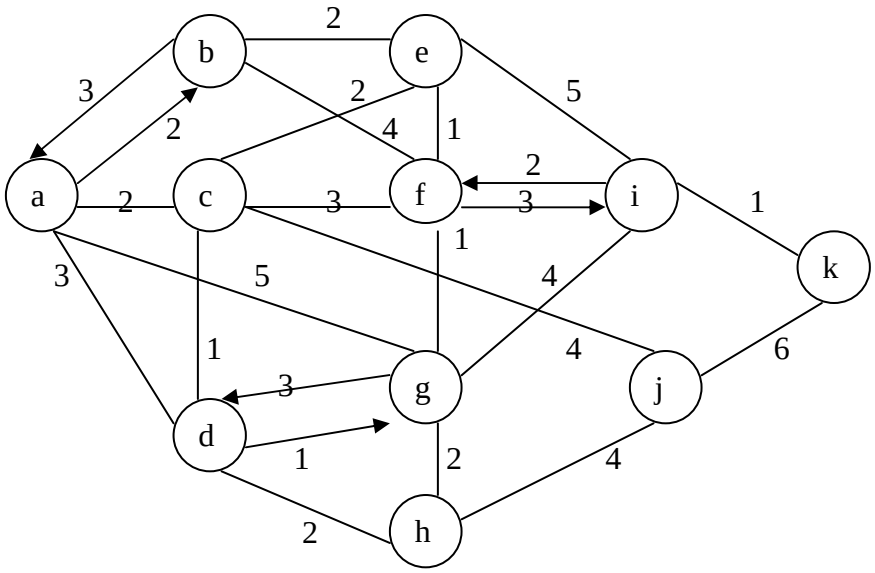


Figura 3.25

Hay muchas rutas diferentes entre a y k, pero lo que se desea es seleccionar la que tenga el menor tiempo, costo, o distancia. Los números que aparecen sobre las flechas de la red pueden representar cualquiera de estas medidas u otras, y la suma de ellas es la que se va a minimizar. Existen diferentes procedimientos para lograrlo, primero se considerara un procedimiento gráfico.

Procedimiento gráfico.

1. Comenzando en el origen, *a,* trácense todos los enlaces por medio de los cuales se puede ir de *a* hacia otro nodo y escríbase sobre ellos la distancia directa desde *a* hacia cada uno de dichos nodos. (Ver la figura)

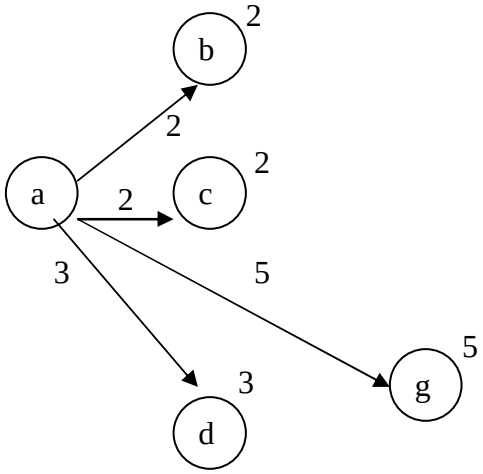


Figura 3.26

2. Si existen conexiones entre cualquiera de los nodos obtenidos en el paso anterior, indíquese para cada una de ellas si la ruta indirecta a partir de *a* es más corta que la directa. Dibújese la más corta como una línea continua y para la más larga utilícese una línea punteada; hágase pasar la distancia más corta encontrada a través de cada nodo. Por ejemplo en la siguiente figura se observa que se puede ir de *a* hacia *q* a través de *d* a un "costo" más bajo que yendo directamente. Además, se puede ir hacia *d* directamente o a través de *c*. En caso de empate dibuje las dos rutas como líneas continuas.

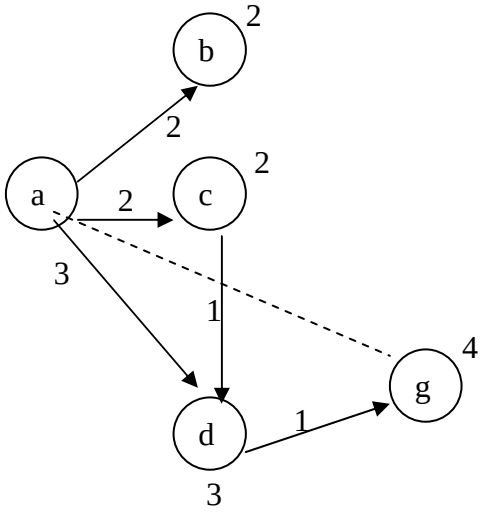


Figura 3.27

3. Agréguense los nodos a los cuales se puede ir desde cualquiera de los obtenidos en el paso 2 y repítase este paso con respecto a ellos; insértense las distancias correspondientes. Este paso se ilustra en la siguiente figura.

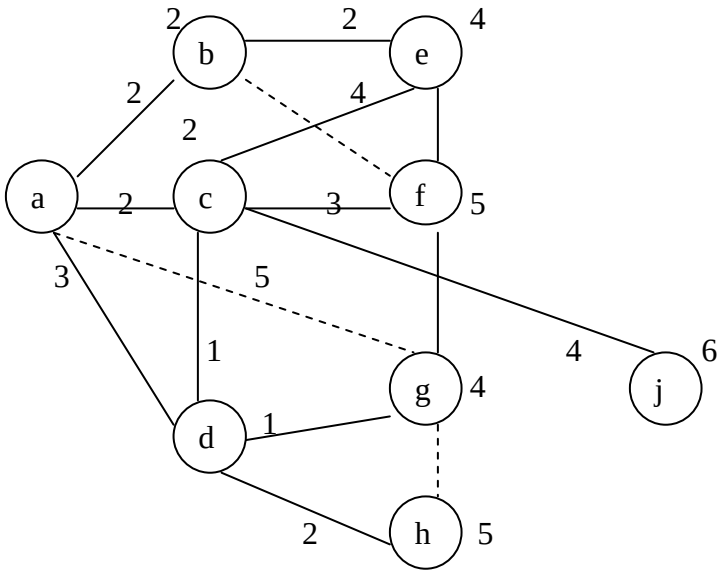


Figura 3.28

4. Continúese así hasta terminar. El diagrama completo aparece en la siguiente figura. Las líneas continuas muestran las rutas que se pueden tomar desde *a* hacia cada uno de los otros puntos. Nótese que hay alternativas. Por ejemplo, se puede ir de *a* hacia *e* a través de *b* ó *c*.

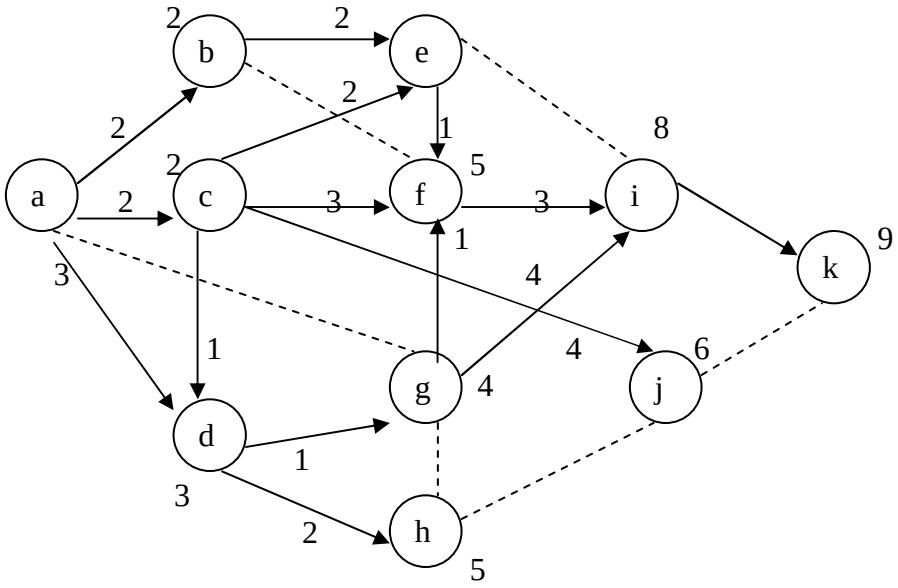


Figura 3.29

Este problema se puede resolver fácilmente sujeto a una restricción adicional, el número de nodos se va a minimizar entre rutas alternativas de igual distancia, tiempo, o costo. Si se impusiera esta restricción al problema que se acaba de resolver, se eliminarían los siguientes enlaces: *cd, ef, gf,* y *gh.*

Procedimiento matricial.

Un segundo método para resolver este problema implica el uso de una "*matriz de dispersión*", que desarrolló Shimbel en 1954. No indica cual es la ruta más corta pero si cual es su longitud. También proporciona la longitud de la ruta más corta entre cualquiera de dos puntos en la red. El procedimiento comprende los siguientes pasos:

1. Transfórmese la red en una matriz estructural de k X k, S = [S_{ij}], donde k es el número de ciudades. Esto se logra seleccionando S_{ij} igual a la distancia de i a j si existe un enlace directo, y S = ∝ si no existe.

   Los elementos de la diagonal S_{ij} todos son ceros. Dicha matriz aparece en la siguiente tabla:

Desde	a	B	c	d	E	f	g	h	i	j	K
A	0	2	2	3	∝	∝	5	∝	∝	∝	∝
B	3	0	∝	∝	2	4	∝	∝	∝	∝	∝
C	2	∝	0	1	2	3	∝	∝	∝	4	∝
D	3	∝	1	0	∝	∝	1	2	∝	∝	∝
E	∝	2	2	∝	0	1	∝	∝	5	∝	∝
F	∝	4	3	∝	1	0	1	∝	3	∝	∝
G	5	∝	∝	3	∝	1	0	2	4	∝	∝
H	∝	∝	∝	2	∝	∝	2	0	∝	4	∝
I	∝	∝	∝	∝	5	2	4	∝	0	∝	1
J	∝	∝	4	∝	∝	∝	∝	4	∝	0	6
K	∝	∝	∝	∝	∝	∝	∝	∝	1	6	0

Figura 3.30 Matriz estructural.

2. Ahora defina una multiplicación especial de dos matrices S y T como sigue:

$$ST = U$$

Donde

$$S = [S_{ij}], \qquad T = [t_{ij}], \qquad U = [u_{ij}]$$

Y

$$u_{ij} = \min \{ S_{i1} + t_{1j} ; S_{i2} + t_{2j} ; S_{i3} + t_{3j} ; \ldots ; S_{ik} + t_{kj} \} = \min_k \{ S_{ik} + j_{kj} \}$$

Utilizando esta regla, multiplíquese S por si misma y obténgase una matriz de dispersión $C = S^2$ Debería estar claro que los elementos de C son las distancias más cortas de i a j en dos o menos pasos. El resultado de este cálculo se muestra en la siguiente tabla:

Desde	a	b	C	d	E	f	g	h	i	j	k
A	0	2	2	3	4	5	4	5	9	6	∝
B	3	0	4	6	2	3	5	∝	7	∝	∝
C	2	4	0	1	2	3	2	3	6	4	10
D	3	5	1	0	3	2	1	2	5	5	∝
E	4	2	2	3	0	1	2	∝	4	6	6
F	5	3	3	4	1	0	1	3	3	7	4
G	5	5	4	3	2	1	0	2	4	6	5
H	5	∝	3	2	∝	3	2	0	6	4	10
I	9	6	5	7	3	2	3	6	0	7	1
J	6	∝	4	5	6	7	6	4	7	0	6
K	∝	∝	10	∝	6	6	5	10	1	6	0

Figura 3.31 Matriz de dispersión.

3. Las potencias de S obedecen las reglas generales de los índices, es decir, S^a X S^b = S^{a+b}, y S^n contiene las distancias más cortas de i a j en n pasos o menos. La ruta más corta de i a j puede contener, cuando más, k – 1 pasos; cualquier ruta conteniendo k pasos debe contener un ciclo y pudiera reducirse. Por lo tanto, los elementos de S^{k-1} deben ser las rutas más cortas. Sin embargo, si $S^n = S^{n-1}$, $n < k - 1$, entonces S^n comprende las rutas más cortas. En vez de calcular las potencias por multiplicaciones sucesivas de S, es más rápido utilizar cuadrados sucesivos y calcular S, S^2, S^4, S^8, ... Las rutas más cortas se habrán encontrado en el paso r cuando $S^{2r} = S^{2(r-1)}$ o cuando $2^r \geq k - 1$ para la primera vez. En nuestro ejemplo $S^8 = S^{16}$; la solución aparece en la tabla de la matriz solución. Si se desea identificar las rutas más cortas, se puede realizar comparando las tablas de la matriz estructural y la matriz de solución. Cualesquiera elementos que sean iguales en ambas matrices constituyen las trayectorias más cortas que deben integrarse a la ruta solución.

En efecto, ninguna ruta más corta crecería si se despreciaran todas las rutas más cortas en un paso, menos una.

En la tabla de Matriz solución se presentan los elementos que son iguales en la tabla de la matriz estructural.

Los elementos no presentados deben ser la suma de dos o más de los presentados. Por ejemplo, la distancia más corta de *a* hacia *e*, que es 4, debe pasar a través de *b*, *c*, ó *d*. Por lo tanto, pudiera ser *abe* ó *ace*.

De esta manera se puede identificar cada una de las distancias más cortas en dos pasos. A partir de éstas se puede construir las distancias más cortas en tres pasos y así sucesivamente, hasta que se identifiquen todas las trayectorias.

Desde	a	b	c	d	E	F	g	H	i	j	k
A	0	2*	2*	3*	4	5	4	5	8	6	9
B	3*	0	4	5	2*	3	4	6	6	8	7
C	2*	4	0	1*	2*	3*	2	3	6	4*	7
D	3*	5	1*	0	3	2	1*	2*	5	5	6
E	4	2*	2*	3	0	1*	2	4	4	6	5
F	5	3	3*	4	1*	0	1*	3	3*	7	4
G	5*	4	4	3*	2	1*	0	2*	4*	6	5
H	5	6	3	2*	4	3	2*	0	6	4*	7
I	7	5	5	6	3	2*	3	5	0	7	1*
J	6	8	4*	5	6	7	6	4*	7	0	6*
K	8	6	6	7	4	3	4	6	1*	6*	0

Figura 3.32 Matriz solución.

El problema del agente viajero es un problema de direccionamiento que está sujeto a restricciones bastante severas, ya que la manera de llegar a un nodo tiene un efecto importante en la manera de salir de él.

La anterior suposición no siempre es válida en los ejemplos estudiados; de cualquier modo para encontrar la ruta más corta entre *a* y *k*, se debe encontrar la ruta más corta desde *a* hasta cada uno de los otros puntos de la red que permitan aproximarse al nodo k, lo cual puede ser muy útil en casos reales.

Existe un procedimiento sencillo de gran utilidad y uso para solucionar el problema del Agente Viajero, será el tema de estudio de la siguiente sección, ya que permitirá ilustrar la manera de solucionar el problema del agente viajero sin restricciones y servirá como método de comparación de rendimiento al realizar las pruebas del algoritmo genético a desarrollar.

3.5 Algoritmo del Mejor Vecino

Para ilustrar este algoritmo se utilizará un ejemplo, para después enunciar formalmente el algoritmo.

Supóngase que nuestro viajante ha de visitar 30 ciudades europeas, debiendo comenzar y finalizar en Madrid.

A continuación se muestran las ciudades con las que se trabajara, mostrando el número por el cual se representarán, así como el grafo que describe el problema, en el cual cada nodo representa una ciudad y cada arista el coste o distancia por carretera de ir de la ciudad i a la ciudad j.

En realidad el grafo es completo ya que en el mundo real, todas las ciudades están relacionadas entre si, pero para que pueda apreciarse de manera más clara, los vértices están ordenados según su situación geográfica y sólo están incluidas las aristas más atractivas, es decir aquellas que unen las ciudades más cercanas entre sí.

También se muestra a continuación una tabla que contiene las distancias que hay entre las distintas ciudades.

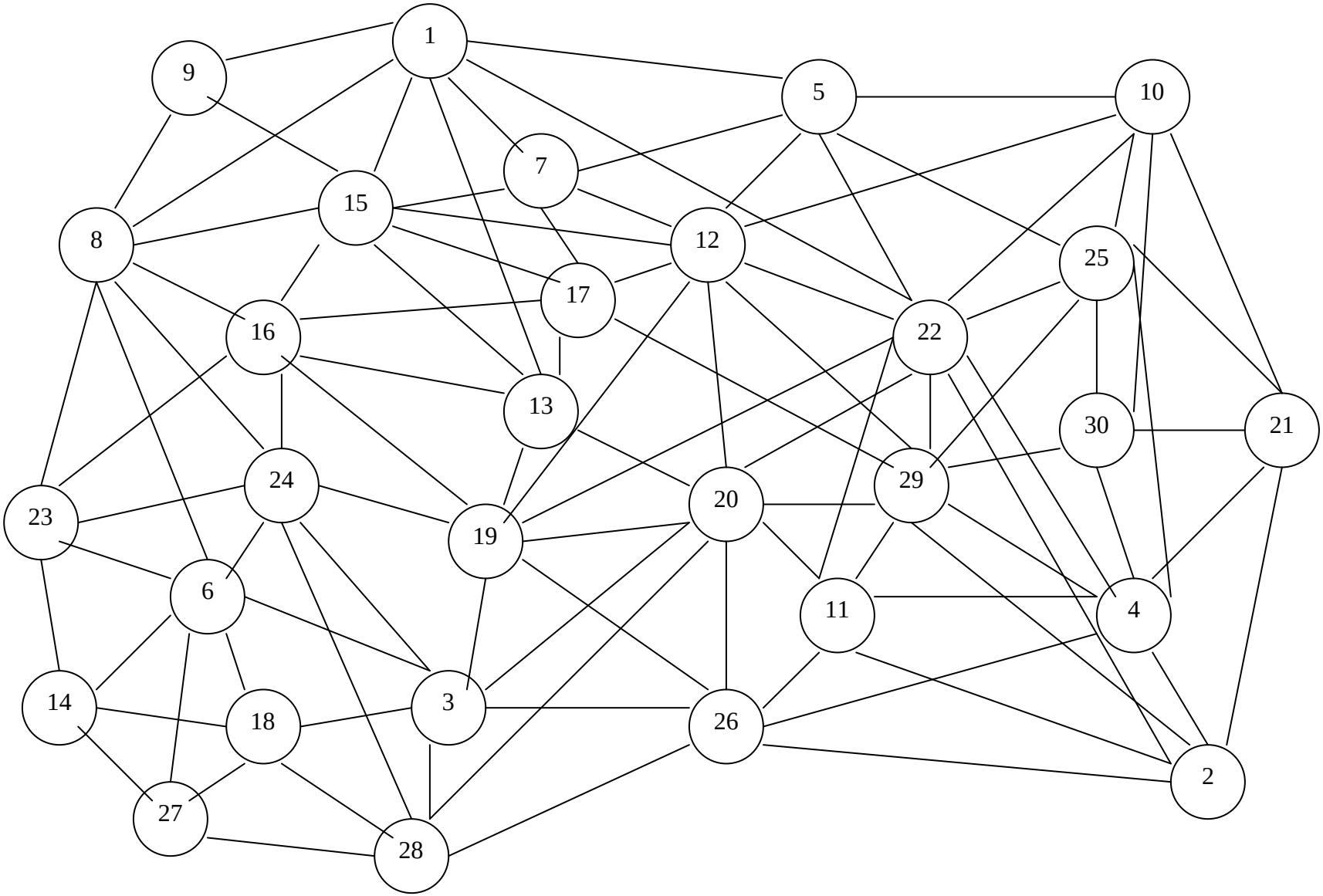
Ciudades

1. Ámsterdam
2. Atenas
3. Barcelona
4. Belgrado
5. Berlín
6. Bilbao
7. Bruselas
8. Dublín
9. Edimburgo
10. Estocolmo
11. Florencia
12. Francfort
13. Ginebra
14. Lisboa
15. Liverpool
16. Londres
17. Luxemburgo
18. Madrid
19. Marsella
20. Milán
21. Moscú
22. Munich
23. Oporto
24. París
25. Praga
26. Roma
27. Sevilla
28. Valencia
29. Venecia
30. Viena

Figura 3.33 Matriz de adyacencia entre las ciudades de Europa

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	∝	2836	1557	1718	665	1422	206	645	743	1386	1370	427	912	2277	516	405	379	1774	1241	1070	2463	831	2064	500	850	1638	2302	1908	1230	1146
2	2836	∝	3090	118	2584	3422	2792	3586	3823	3568	2115	2396	2446	4320	3504	3252	2637	3760	2621	2128	3169	2063	4141	2912	2154	2389	4117	3405	1878	1862
3	1557	3090	∝	1972	1860	620	1360	912	2059	2699	1079	1326	749	1235	1799	1442	1151	621	500	977	3630	1345	1084	1037	1706	1347	1046	349	1231	1797
4	1718	118	1972	∝	1466	2304	1674	2468	2705	2450	997	1278	1328	3202	2386	2134	1519	2642	1503	1010	2194	945	3023	1794	1036	1271	2999	2317	760	744
5	665	2584	1860	1466	∝	1970	774	1447	1595	1001	1207	538	1076	2825	1335	977	767	2321	1544	1033	1829	586	2611	1048	345	1475	2850	2210	1043	636
6	1422	3422	620	2304	1970	∝	1233	1794	1942	2745	1432	1488	1102	895	1682	1324	1269	395	853	1330	3784	1747	682	920	1942	1700	933	633	1584	2143
7	206	2792	1360	1674	774	1233	∝	714	862	1512	1188	386	715	2087	601	244	211	1583	1044	888	2575	731	1873	310	894	1456	2112	1710	1142	1105
8	645	3586	1912	2468	1447	1794	714	∝	379	2202	1905	1102	1329	2648	167	456	999	2145	1658	1605	3412	1545	2435	871	1611	2173	2673	2261	1858	1822
9	743	3823	2059	2705	1595	1942	862	379	∝	968	2052	1250	1477	2796	363	635	1147	2293	1806	1752	3649	1692	2583	1019	1759	2320	2821	2409	2006	1969
10	1386	3568	2699	2450	1001	2745	1512	2202	968	∝	2176	1407	1992	3608	2089	1732	1535	3096	2383	2032	1292	1556	3386	1823	1349	2444	1706	3050	2012	1640
11	1370	2115	1079	997	1207	1432	1188	1905	2052	2176	∝	974	602	2302	1792	1435	979	1682	611	296	2800	617	2074	1131	1005	267	2082	1430	255	821
12	427	2396	1326	1278	538	1488	386	1102	1250	1407	974	∝	590	2342	990	632	236	1839	1010	674	2360	398	2129	576	502	1242	2329	1677	855	713
13	912	2446	749	1320	1076	1102	715	1329	1477	1992	602	590	∝	1971	1216	859	506	1352	433	317	2898	595	1743	501	922	871	1752	1099	579	989
14	2277	4120	1235	3202	2825	895	2087	2648	2796	3600	2302	2342	1971	∝	2536	2179	2124	620	1723	2199	4682	2568	304	1775	2797	2570	406	970	2454	3020
15	516	3504	1799	2386	1385	1682	601	167	363	2089	1792	990	1216	2536	∝	343	886	2032	1546	1492	3330	1432	2322	759	1498	2060	2561	2149	1746	1709
16	405	3252	1442	2134	977	1324	244	456	635	1732	1435	632	859	2179	343	∝	529	1675	1188	1135	3078	1075	1965	401	1141	1703	2204	1791	1388	1352
17	379	2637	1151	1519	767	1269	211	999	1147	1535	979	236	506	2124	886	529	∝	1620	835	679	2561	515	1910	357	728	1247	2154	1502	933	928
18	1774	3760	621	2642	2321	395	1583	2145	2293	3096	1682	1839	1352	620	2032	1675	1620	∝	1103	1580	4172	1948	561	1272	2309	1950	538	352	1834	2400
19	1241	2621	500	1503	1544	853	1044	1658	1806	2383	611	1010	433	1723	1546	1188	835	1103	∝	508	3311	999	1495	774	1390	879	1503	851	763	1329
20	1070	2128	977	1010	1033	1330	688	1605	1752	2032	296	674	317	2199	1492	1135	679	1580	508	∝	2817	495	1971	845	870	564	1980	1327	273	839
21	2463	3169	3630	2194	1829	3784	2575	3412	3649	1292	2800	2360	2898	4682	3330	3078	2561	4172	3311	2817	∝	2362	4503	2863	1860	3074	4657	3975	2563	1965
22	831	2063	1345	945	586	1747	731	1545	1692	1556	617	398	595	2568	1432	1075	515	1948	999	495	2362	∝	2386	833	384	885	2349	1696	453	394
23	2064	4141	1084	3023	2611	682	1873	2435	2583	3386	2074	2129	1743	304	2322	1965	1910	561	1495	1971	4503	2386	∝	1561	2584	2342	622	912	2226	2784
24	500	2912	1037	1794	1048	920	310	871	1019	1823	1131	576	501	1775	759	401	357	1272	774	845	2836	833	1561	∝	1029	1400	1800	1388	1108	1230
25	850	2154	1706	1036	345	1942	894	1611	1759	1349	1005	502	922	2797	1498	1141	728	2309	1390	870	1860	384	2584	1029	∝	1273	2709	2056	823	291
26	1638	2389	1347	1271	1475	1700	1456	2173	2320	2444	267	1242	871	2570	2060	1703	1247	1950	879	564	3074	885	2342	1400	1273	∝	2350	1698	523	1089
27	2302	4117	1046	2999	2850	933	2112	2673	2821	1706	2082	2329	1752	406	2561	2204	2154	538	1503	1980	4657	2349	622	1800	2709	2350	∝	697	2235	2801
28	1908	3405	349	2317	2210	633	1710	2261	2409	3050	1430	1677	1099	970	2149	1791	1502	352	851	1327	3975	1696	912	1388	2056	1698	697	∝	1582	2148
29	1230	1078	1231	760	1043	1584	1142	1858	2006	2012	255	855	579	2454	1746	1388	933	1834	763	273	2563	453	2226	1108	823	523	2235	1582	∝	583
30	1146	1862	1797	744	636	2143	1105	1822	1969	1640	821	713	989	3020	1709	1352	928	2400	1329	839	1965	394	2784	1230	291	1089	2801	2148	583	∝

Figura 3.34 Grafo representativo del problema de las ciudades europeas.



El algoritmo del mejor vecino consiste en partir de una ciudad inicial, en este caso Madrid y buscar de entre todas las demás ciudades aquella que minimiza la distancia hacia ésta, uniéndolas entre sí. A continuación, busca cual es la ciudad entre las restantes que minimiza la distancia de ir desde ella a la ciudad que ha sido unida con Madrid, uniendo estas dos ciudades entre sí y así seguiría sucesivamente hasta que ya hubieran sido incluidas todas las ciudades, el algoritmo termina cuando la última ciudad quede unida con Madrid y quede conformando el tour que será la solución del problema.

El resultado final indica el siguiente recorrido:

18, 28, 3, 19, 13, 20, 29, 11, 26, 22, 25, 30, 5, 12, 17, 7, 1, 16, 15, 8, 9, 10, 21, 4, 2, 24, 6, 23, 14, 27, 18.

Costo total: 17761 Km.

En realidad se trata de construir un ciclo Hamiltoniano de bajo coste basándose en el vértice más cercano a uno dado. Este algoritmo se debe a Rosenkrantz, Stearns y Lewis (1977) y puede describirse en los siguientes pasos:

```
Inicio
t, j vértices
V(j) arreglo de vértices a visitar
Seleccione un vértice j al azar
t = j
Eliminar j del arreglo V(j)
Mientras el arreglo V(j) tenga elementos
        Seleccionar de V(j) el vértice j tal que la distancia de t a j sea la mínima
        Conectar t a j
        t = j
        Eliminar j del arreglo V(j)
Fin mientras
Fin
```

Si se sigue la evolución del algoritmo a través del ejemplo se vera que comienza muy bien, seleccionando aristas de bajo costo. Sin embargo al acercarse al final del proceso probablemente quedarán vértices cuya conexión obligará a introducir aristas de coste elevado. A este problema se le conoce como "*miopía"* del algoritmo, ya que, en una iteración se elige la mejor opción sin "*ver"* que esto puede obligar a realizar malas elecciones en iteraciones posteriores.

El algoritmo como tal se puede programar en unas pocas líneas de código. Sin embargo una implementación directa será muy lenta al ejecutarse sobre ejemplos de gran tamaño (1000 ciudades). Así pues, incluso para un heurístico tan sencillo como éste, es importante pensar en la eficiencia y velocidad de su código.

Para reducir la miopía del algoritmo y aumentar su velocidad se puede introducir el concepto de "*subgrafo candidato*", junto con algunas modificaciones en la

exploración. Un subgrafo candidato es un subgrafo del grafo completo con los n vértices y únicamente las aristas consideradas "*atractivas*" para aparecer en un ciclo Hamiltoniano de bajo coste. Una posibilidad es tomar, por ejemplo, el subgrafo de los k vecinos más cercanos; esto es, el subgrafo con n vértices y para cada uno de ellos las aristas que lo unen con los k vértices más cercanos.

El algoritmo puede mejorarse en los siguientes aspectos:

- Para seleccionar el vértice j que se va a unir a t (y por lo tanto al tour parcial en construcción), en lugar de examinar todos los vértices, se examinan únicamente los adyacentes a t en el subgrafo candidato. Si todos ellos están ya en el tour parcial, entonces sí que se examinan todos los posibles.
- Cuando un vértice queda conectado (con grado 2) al tour en construcción, se eliminan del subgrafo candidato las aristas incidentes en él.
- Se especifica un número s < k de modo que cuando un vértice que no está en el tour está conectado únicamente a s o menos aristas del subgrafo candidato se considera que se ésta quedando aislado. Por ello se inserta inmediatamente en el tour. Como punto de inserción se toma el mejor de entre los k vértices más cercanos presentes en el tour.

Con este panorama del problema del agente viajero y sus posibles algoritmos de solución se puede ahora proponer un algoritmo genético para solucionar el problema; en ello consiste el tema del siguiente capítulo.

CAPÍTULO IV

DESARROLLO DEL MODELO

En el capítulo tres se presentó el algoritmo genético simple, el cual posee características muy particulares dentro de un esquema que permite, en principio, múltiples variantes. Un algoritmo genético se caracteriza por utilizar un esquema de selección, un esquema de cruzamiento de un solo o varios puntos de corte, mutaciones distribuidas uniformemente en la población y que operan con la misma probabilidad para todos los individuos durante todas las generaciones en que el algoritmo genético es ejecutado. También se ha presentado el esquema de codificación del domino del problema basado en cadenas binarias de una cierta longitud.

No solo es cierto que no tiene por que ser siempre así, sino que además, en muchos casos no es conveniente que lo sea. En este capítulo se presentará el modelo del algoritmo genético que se aplicará para la solución del problema del agente viajero y con él las diversas características que se modificaron para complementar o reemplazar a las antes ya mencionadas.

4.1 Criterio de optimización.

En muchos problemas que se resuelven por algoritmos genéticos se tiene una función de evaluación única e invariable para todas las generaciones del algoritmo genético. Pero en el caso del agente viajero, ¿cuál debe ser la función de evaluación de cada individuo?

Se pretende minimizar el costo de un viaje redondo que pase por todas las ciudades. Desde aquí comienzan los problemas, ya que hasta ahora se han considerado problemas de maximización que por su naturaleza están adaptados para los algoritmos genéticos, además es bien sabido que todo problema de minimización puede ser traducido a uno de maximización simplemente multiplicando por –1 la función de evaluación. Así al maximizar ésta, se minimiza implícitamente la función objetivo. Pero en el caso del agente viajero, dado que todos los costos son no negativos, al multiplicar por –1 se obtendrán valores negativos no permitidos por el esquema de selección de los algoritmos genéticos. Una alternativa podría ser que el valor de adaptación de cada individuo sea la diferencia entre su costo total y el costo máximo posible de un ciclo en la instancia del problema que se pretende resolver. Pero esto significa que se debe conocer el costo máximo de un ciclo en dicho problema. Esto es tan difícil como conocer el costo del ciclo mínimo, así que en este caso no es útil esta propuesta.

Otra opción consiste en cambiar el esquema de selección por uno en que la bondad de un individuo no sea proporcional al valor obtenido mediante la función de evaluación.

Un esquema que se puede usar en este caso es la selección conocida en inglés como "*linear ranking*", es este esquema, la probabilidad de elección de un individuo está dada por su posición (ranking) respecto al total de la población ordenada.

Para aplicar este esquema se debe ordenar la población de manera creciente en función de la longitud del ciclo que representa cada individuo, y luego se seleccionan los individuos asignándoles mayor probabilidad de ser elegidos cuanto más cercanos se encuentren del principio de la lista.

Otra opción consiste en utilizar un remapeo de la función objetivo (costo de viaje) para obtener una función de adaptación. Un remapeo útil para el agente viajero es el siguiente: sean f_{max}, f_{min} y f_i el costo máximo, mínimo y el del i-ésimo individuo en alguna población, respectivamente, la función de adaptación evaluada en el i-ésimo individuo es:

$$\text{Adap (i)} = (f_{max} + f_{min}) - f_i$$

Haciendo esto ya es posible utilizar el esquema de selección proporcional.

Así que el criterio de optimización será encontrar la ruta más corta en la lista del "linear ranking" en un número de generaciones que será establecido por el usuario.

4.2 Representación.

El siguiente problema a resolver es el de la codificación del dominio de soluciones, como se ha explicado anteriormente la marcada preferencia por usar representaciones binarias en los algoritmos genéticos se deriva de la teoría del esquema, la cual trata de analizar los algoritmos genéticos en función de su comportamiento esperado al muestrear esquemas. El argumento fundamental para justificar el fuerte énfasis en los alfabetos binarios se deriva del punto de vista de procesamientos de esquemas, por el hecho de que si se usa un alfabeto binario, el número de esquemas es máximo para un número finito de puntos de búsqueda. Consecuentemente la teoría del esquema favorece la representación binaria de las soluciones. Pero los algoritmos genéticos no están restringidos a genomas codificados en binario. De hecho, existen casos considerables en que un genoma codificado en binario no es conveniente, y uno de estos casos es el problema del agente viajero, como ya hemos mencionado en el capítulo anterior el problema del agente viajero se trata de encontrar la trayectoria cíclica más corta que debe seguir el agente a través de todas las ciudades de un conjunto predefinido, visitando cada ciudad sólo una vez. Las distancias entre las ciudades puede expresarse fácilmente con una matriz de adyacencias como la que se muestra a continuación.

	A	B	C	D	E
A	0	20.4	22.2	29.3	17.4
B	20.4	0	29.7	21.8	16.7
C	22.2	29.7	0	19.3	12.6
D	29.3	21.8	19.3	0	11.9
E	17.4	16.7	12.6	11.9	0

Figura 4.1 Matriz de distancias para el problema del agente viajero.

Número	Ruta	Distancia
1	CEDBA	66.7
2	DECAB	67.1
3	CDEBA	68.3
4	BDECA	68.5
5	DCEBA	69.0
6	DCEAB	69.7
7	BEDCA	70.1
8	BDCEA	71.1
9	DEBAC	71.2
10	DBAEC	72.2

Figura 4.2 Las mejores 10 rutas para el problema del agente viajero.

1 ciudad	2 ciudad	3 ciudad	4 ciudad	5 ciudad
100	011	001	000	010

Figura 4.3 Una posible codificación para el problema del agente viajero.

Aún en esta simple instancia puede apreciarse la complejidad computacional inherente al problema. Hay n! rutas posibles (en el ejemplo n = 5, es decir 5X4X3X2 = 120 rutas). Aún si se considera que hay, de hecho, la mitad de trayectorias (debido a que toda trayectoria es equivalente a su inversa, por ejemplo ABCDE = EDCBA) por examinar, la enumeración exhaustiva sólo es posible cuando n es pequeña. De hecho no se conoce un algoritmo determinístico que resuelva el problema en un tiempo que dependa polinomialmente de n para cualquier n (lo que como se estudio anteriormente hace de esté un problema NP-completo).

¿Cómo puede abordarse este problema con un algoritmo genético? La respuesta podría consistir en proceder de la siguiente manera:

a) Codificar cada ciudad como un número binario.
b) Considerar que una solución es la concatenación de tantos números como ciudades existan.
c) Generar un conjunto de tales candidatos a solución.
d) Aplicar un algoritmo genético.

Posibles individuos de tal población se muestran en la figura 4.3.

Surgen inmediatamente dos problemas. El primero es obvio: no puede permitirse que una ciudad, por definición, se repita en el genoma; ni puede haber números de ciudades fuera del rango especificado. En este caso, las combinaciones binarias 101, 110 y 111 están prohibidas. Por lo tanto, debe revisarse la suposición de que los elementos de la población inicial son generados aleatoriamente. Debe garantizarse un conjunto inicial de individuos correctos. El segundo problema es un poco más sutil pero se relaciona con el primero. Aún forzando a la población inicial para que cumpla con las dos restricciones anteriores, se debe ser cuidadoso al aplicar cruzamiento y mutación. Los genomas resultantes pueden inducir cadenas con ciudades repetidas o con ausencias.

¿Cómo enfrentamos estos dos problemas? Una posibilidad es aplicar un castigo a los individuos incorrectos. Con esto se espera disuadir al algoritmo de proponer los esquemas equivocados. En este caso, parecería que el algoritmo genético podría gastar mucho tiempo (y de hecho lo hace) mientras aprende que las combinaciones prohibidas no son deseables.

Otra opción es aplicar un algoritmo reparador tal que el orden correcto es restaurado después de aplicar los operadores genéticos. En este caso no parece intuitivo que, cuando se reparan los genomas incorrectos, no se perderá parte de la información que ha incorporado al proceso genético.

Finalmente, puede dejarse la codificación binaria de lado y trabajar sobre una base diferente. Sin embargo, cuando se trabaja con una base diferente de dos se cae en un tipo diferente de problema. El principal es ¿cómo realizar el cruce? y por supuesto ¿cómo se muta? Se responderá a estas cuestiones en las siguientes secciones, primeramente se analizara como implementar la representación.

La manera más natural de representar a los individuos de una población de posibles soluciones para el problema del agente viajero es asignar un identificador (un carácter o un número entero) a cada ciudad, establecer que en cada posición de la cadena genética que constituye un individuo debe colocarse uno de estos identificadores, es decir, cada alelo es un identificador (en representación binaria cada alelo sólo podía ser 0 ó 1). Se establecen las restricciones de que en un individuo debe aparecer cada identificador de ciudad una y sólo una vez, y que todos los individuos tienen el mismo primer alelo (es decir todos los ciclos representados en una población comienzan en la misma ciudad).

En estas condiciones es sencillo asegurar que la población inicial sólo contendrá cadenas válidas.

Supóngase que n denota el número de ciudades. Se aplicara el siguiente algoritmo utilizando como identificador único para cada ciudad una letra del abecedario.

Para i desde 1 hasta N (donde N es el tamaño de la población)
Hacer una lista L de los n identificadores de ciudad distintos
Colocar el primer identificador en la primera posición del código genético del i-ésimo individuo (primer alelo)
Eliminar el primer identificador de la lista L

Para cada posición j del código genético del individuo, desde 2 hasta n
Elegir aleatoriamente un identificador de L
Colocar ese identificador en la posición j del código del individuo i
Eliminar el identificador de L

Este algoritmo proporciona una población inicial de N cadenas válidas para el problema del agente viajero. Todas ellas denotan un ciclo posible, no se repiten ni faltan ciudades y todos los ciclos comienzan en la misma ciudad. En este caso la representación no es binaria, aún cuando finalmente en la memoria de la computadora todos los identificadores de ciudad serán almacenados en binario, para el algoritmo genético cada identificador constituye un alelo, un ente indivisible que no puede ser partido ni alterado internamente, el análogo de cada bit en las anteriores representaciones. Un ejemplo se muestra en la siguiente figura.

A	B	D	C	E	Individuo1
A	D	E	C	B	Individuo2
A	E	B	D	C	Individuo3

Figura. 4.4 Tres posibles individuos codificados en letras.

Ahora se debe asegurar que los operadores genéticos mapeen individuos válidos a partir de individuos válidos.

4.3 Criterio de selección.

En este caso se utiliza el esquema del “linear ranking”, pero alterando además el esquema de selección por medio de la técnica de elitismo, la que en esté caso consistirá en seleccionar a las mejores soluciones de la lista para pasarlas a la siguiente generación y así preservar este importante material genético.

Una vez separados estos individuos, se aplica como método de selección la “ruleta” (roulette wheel seleccion), primeramente se analiza la ruleta, su funcionamiento y a mas adelante en el capítulo se explica la necesidad de uso de una función de remapeo para la función de adaptación como se estudiará en el ejemplo final del capítulo.
La selección de Ruleta, también conocida como “Rueda de la ruleta “(Holland) tiene el siguiente funcionamiento.

Los padres se seleccionan de acuerdo a su función de aptitud, mientras mejor puntuación tiene un individuo tendrá mayor posibilidad de ser seleccionado.

Supóngase una ruleta en la que el 100% de la circunferencia representa el total de la suma de las puntuaciones de aptitud de todos los individuos de la población, luego a cada individuo se le asigna el trozo que le corresponde de ésta, según su aportación a el total de la suma de las calificaciones.

Esto se logra por medio del siguiente procedimiento:

Calcular la puntuación de cada cromosoma de acuerdo a su función de aptitud

eval(V_i)=(i=1 … hasta n) donde n es el tamaño de la población.

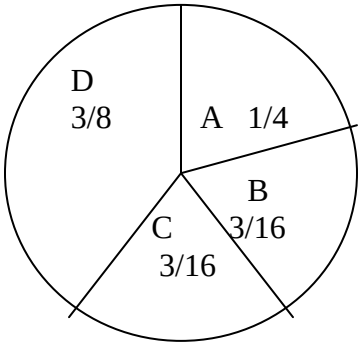
Calcular la aptitud total de la población F

$$F = \sum_{i=1}^{n} eval(V_i)$$

Es decir, si la calificación de un individuo es V_i entonces le corresponde un segmento de circunferencia dado por una simple regla de tres:

$$P_i = \frac{eval(V_i)}{F}$$

¿Qué ocurrirá entonces si se considera ésta como una ruleta y se coloca una lengüeta que roce el borde de ella? La probabilidad de que dicha lengüeta quede en el arco correspondiente al individuo V_i de calificación eval(Vi) cuando la ruleta se detenga tras realizar algunos giros será proporcional a la probabilidad P_i del individuo. Como lo ilustran las siguientes figuras:



Genotipo	Calificación
A	4
B	3
C	3
D	6
Total	16

Figura 4.5 Selección por ruleta.

4.4 Cruce

Teniendo el esquema de codificación descrito antes basado en letras y el método que genera una población inicial válida. Si bajo este esquema se utilizara el cruzamiento de un punto, comenzarían a surgir individuos no válidos en la población. Supóngase que se tienen dos individuos válidos con las rutas ACDBE y ADECB respectivamente, ahora supóngase que utilizando cruzamiento de un punto, se decide cortar ambas cadenas entre los alelos 3 y 4, se generarían entonces las cadenas descendientes: ACDCB y ADEBE, ambas inválidas dado que repiten ciudades. ¿Qué se puede hacer en este caso? Existen cuatro opciones:

- Una es la ya comentada de castigar a cada cadena errónea haciendo que tenga una calificación muy baja por violar las reglas.
- Corregir cada cadena errónea mapeándola en a una válida.
- Cambiar el mecanismo de codificación de tal forma que los operadores genéticos conocidos no violen las reglas.
- Y la última y más barata computacionalmente hablando, consiste en diseñar operadores genéticos de cruza y mutación que generen siempre cadenas válidas.

La última será la opción elegida, ya que es posible diseñar operadores genéticos de cruza y mutación que siempre generen cadenas válidas. Comencemos con el operador de cruza que se encargará de esto, el cruzamiento uniforme ordenado.

Supóngase que se poseen dos cadenas genéticas P_1 y P_2, es decir, dos individuos de la población que denotan dos ciclos diferentes en la gráfica de las ciudades. Una manera de generar dos individuos nuevos a partir de estos dos es la siguiente:

Sea M una cadena binaria de longitud en bits igual a la longitud de los individuos, es decir el número de ciudades. Por cada bit de M se lanza una moneda al aire y si cae cara se pone un 1 en la posición correspondiente de M, en otro caso se pone un 0. Este proceso genera una máscara M con, aproximadamente, el mismo número de 1's que de 0's. Sea H una cadena genética vacía, se copia en H las posiciones de P_1 que corresponden a 1's en M. Para que H sea una cadena válida debe incluir aquellas ciudades de P_1 que corresponden a 0's en M, así que se procede ahora a colocarlas en H, pero en el orden en que aparezcan en P_2. Así se genera H que visita al mismo tiempo algunas ciudades que su padre P_1, y el resto son visitadas en el mismo orden que su padre P_2.

El algoritmo que realiza el proceso es el siguiente:

Algoritmo del cruzamiento uniforme.

1. Se seleccionan dos padres P_1 y P_2 de la población. Sea I la longitud de dichos padres (es decir el número de ciudades).

2. Se genera una máscara M con longitud de I bits, cada bit de la cadena se genera aleatoriamente, 0 y 1 son equiprobables.
3. Sea F una cadena genética vacía y la variable *faltan* = 1.
4. Por cada posición i de una cadena genética H (descendiente de P_1 y P_2) inicialmente vacía:

Si M[i] = 1 entonces
    H[i] = P_1[i]
si no
    F[faltan] = P_1[i]
    faltan = faltan + 1

1. Permutar F de tal forma que aparezcan las ciudades en el mismo orden en que aparecen en P_2.
2. Sea k = 1
3. Por cada posición j vacía de H
    H[j] = F[k]
    k = k +1

Por supuesto la mejor manera de entender este procedimiento es mediante un ejemplo:

Las siguientes figuras ilustran un ejemplo de este tipo de cruzamiento.

A	B	D	C	E	Individuo 1
A	D	E	C	B	Individuo 2
1	0	1	1	0	Máscara
A		D	C		Hijo 1 (se copia del ind 1, faltan B y E)
A		E	C		Hijo 2 (se copia del ind 2, faltan D y B)

A	B	D	C	E	Individuo 1
A	D	E	C	B	Individuo 2
1	0	1	1	0	Máscara
A	E	D	C	B	Hijo 1 (E aparece antes que B en ind 2)
A	B	E	C	D	Hijo 2 (B aparece antes que D en ind 1)

Figura 4.6 Los dos pasos para generar un par de descendientes de una pareja de individuos con cruzamiento uniforme ordenado.

4.5 Mutación

Como ya se menciono si se utiliza el operador de mutación tradicional en el problema del agente viajero con la codificación ya descrita, al igual que en el caso del cruzamiento, se pueden generar cadenas inválidas. Pero también es posible definir un nuevo operador de mutación que asegure que siempre que se altere una cadena válida el resultado sea otra cadena válida. Se procederá de manera similar al caso del cruzamiento.

1. Sea P_1 una cadena genética de longitud *l*

2. Sea M una cadena binaria de longitud *l* y p la probabilidad de mutación.

3. Para cada posición i de M desde 2 hasta *l* (se excluye la primera posición) se elige aleatoriamente el valor de M[i], con probabilidad p se elige 1, con probabilidad 1-p se elige 0.

4. Si el número M tiene exactamente un 1 en la posición r, entonces se elige aleatoriamente otra posición de M distinta de 1 y de r para colocar otro 1.

5. Sea Temp una cadena genética inicialmente vacía y cont = 1

6. Por cada posición j de P_1 desde 2 hasta *l*
   Si M[j] = 1 entonces
   Temp.[cont] = P_1[j]
   cont = cont + 1
   P_1[j] = vacío

7. Permutar los elementos de Temp. de tal forma que ninguno ocupe su posición original en Temp.

8. k = 1

9. Para cada i desde 1 hasta *l*
   Si P_1 [i] = vacío entonces
   P_1 [i] = Temp.[k]
   k = k + 1

Con este algoritmo se asegura que las cadenas generadas por mutación a partir de cadenas válidas son también válidas, dado que únicamente se altera el orden en que son visitadas las ciudades, aunque cabe aclarar que, dado lo que se especifica en el paso 4, la probabilidad de mutación ya no es p.

A pesar de que él análisis se ha centrado en el problema del agente viajero, las conclusiones son aplicables a una amplia gama de problemas en los que pueden presentarse todas o algunas de las condiciones analizadas.

Se presenta ahora un ejemplo realizado a mano de este algoritmo a fin de ilustrar de mejor manera este procedimiento si bien debido al costo de los cálculos manuales sólo se muestran algunos de los aspectos fundamentales del proceso de creación de una nueva generación.

Se considera la siguiente situación: Un problema del agente viajero de sólo cinco ciudades, representadas por las letras A, B, C, D, y E y con las distancias entre ellas representadas por la siguiente matriz de adyacencia, donde para facilitar las operaciones se han elegido como distancias múltiplos de 10.

	A	B	C	D	E
A	0	10	30	20	40
B	10	0	20	10	30
C	30	20	0	20	30
D	20	10	20	0	10
E	40	30	30	10	0

Figura 4.7 Matriz de adyacencia.

Se comienza con el proceso de generar una población inicial que en este caso limitado será de 10 individuos.

Siguiendo el algoritmo expuesto anteriormente en el capítulo:

i contador, n tamaño de la población, P_i individuo generado y *l* lista de ciudades.

Para i = 1 hasta n = 5
    l = { A, B, C, D, E }
    P_1 = [A,,,,] // Primer alelo igual al primer elemento de *l*
    l = { B, C, D, E } // Se elimina el primer elemento de *l*
    Para j = 2 hasta n
    Se elige el siguiente elemento de P_1 aleatoriamente de los restantes
    miembros de *l*
    P_1 = [A,D,,,]
            Se elimina el miembro elegido de *l*
            l = { B, C, E }
    Next j
Next i

Se terminan todas las cadenas con el mismo alelo con el que iniciaron es decir A, por lo que no es necesario escribirlo para efecto de facilitar la representación y las posteriores operaciones genéticas de cruce y mutación.

Continuando este proceso se genera toda una población de cromosomas válidos que comienzan en el mismo alelo y permiten iniciar el algoritmo genético.

Población inicial:

Individuos	Representación	Distancia
1	ABCDE	100
2	ABDEC	90
3	ACEBD	120
4	ADCEB	110
5	ABDCE	110
6	AEBCD	130
7	ACBDE	110
8	ADECB	90
9	ACDEB	100
10	ABEDC	100

Figura 4.8 Lista de cromosomas.

Con esta línea es posible realizar la lista del linear ranking la cual consiste en ordenar los cromosomas de forma ascendente con base a las distancias quedando como sigue la lista:

Cromosoma	Distancia
ABDEC	90
ADECB	90
ABCDE	100
ACDEB	100
ABEDC	100
ADCEB	110
ABDCE	110
ACBDE	110
ACEBD	120
AEBCD	130

Figura 4.9 Linear ranking

Ahora de acuerdo con él algoritmo creado, antes de realizar la selección por ruleta, se procede a seleccionar por elitismo los mejores cromosomas para pasarlos a la siguiente generación de manera automática, en este caso se eligen los dos cromosomas ABDEC y ADECB ambos con una distancia de 90 para asegurar que su material genético perdure. Cabe señalar que dichos cromosomas no se eliminan del linear ranking para permitir que su material genético perdure y participe en el proceso de selección, cruza y mutación que darán origen a la nueva generación.

Así se tiene que en la segunda generación habrá 10 individuos, dos de ellos se eligieron por elitismo, de los ocho restantes siete serán generados por selección y cruce mientras que el último será generado por mutación.
A continuación se ejemplifica el proceso de la siguiente manera. Primeramente se suma la columna de distancias del linear ranking y se obtiene 1060 que será la circunferencia total de la ruleta, pero antes se realiza el necesario remapeo de la función de adaptación ya que si se asignan por regla de tres las probabilidades de ser seleccionado en la siguiente generación ocurriría lo siguiente: Observe el caso de dos cromosomas en casos opuestos, los cromosomas ABDEC con distancia 90 y el cromosoma AEBCD con distancia 130, es decir el mejor y peor individuos de la generación, considerando 1060 como el 100 % y tomando el aporte de cada cromosoma a este total para obtener su porcentaje se tiene que:

Para el cromosoma ABDEC $\frac{90}{1060} = \frac{X}{100}$ da X = 8.49 %

Y para el cromosoma AEBCD $\frac{130}{1060} = \frac{X}{100}$ da X = 12.26 %

Es decir el peor individuo recibe el mejor porcentaje mientras que el mejor recibe el menor porcentaje, ¿Cómo corregir esto? Con una función de remapeo como se ha mencionado anteriormente.

El esquema es sencillo obsérvese:

Sea Adap(i) la función de adaptación que será igual a $(f_{max} + f_{min}) - f_i$ donde f_{max}, f_{min} y f_i son la mayor, menor distancia y la distancia del i-ésimo individuo respectivamente.

Así en esté caso se tiene que $f_{max} = 130$ y $f_{min} = 90$ por lo que para los dos cromosomas de ejemplo se tiene:

Para Adap(ABDEC) = (130 + 90) – 90 = 130

Y para Adap(AEBCD) = (130 + 90) – 130 = 90

Por lo que usando estos resultados en la regla de tres la probabilidad de selección es de:

ABDEC = 12.26 % para el mejor cromosoma y

AEBCD = 8.49 % para el peor cromosoma

Lo que va muy de acuerdo con las necesidades del problema, a continuación se muestra la tabla con el resultado de la probabilidad de selección para cada cromosoma de acuerdo a la función de adaptación.

Cromosoma	Distancia	Adaptación %
ABDEC	90	12.26
ADECB	90	12.26
ABCDE	100	11.32
ACDEB	100	11.32
ABEDC	100	11.32
ADCEB	110	10.37
ABDCE	110	10.37
ACBDE	110	10.37
ACEBD	120	9.43
AEBCD	130	8.49

Figura 4.10 Tabla de cromosomas con probabilidad de selección.

Así considerando el 1060 del total de distancias como la circunferencia total de la ruleta se puede visualizar así:

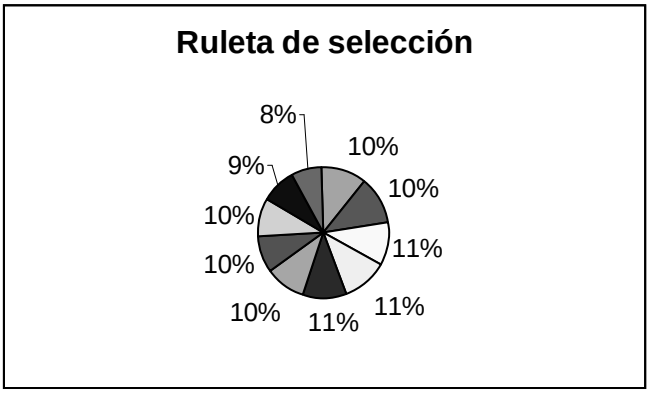


Figura 4. 11 Ruleta de selección basada en la función adaptación.

Ahora obsérvese un sencillo ejemplo de cruce entre dos cromosomas seleccionados.

Suponiendo que los cromosomas ADCEB con coste de 110 y el cromosoma ABDCE con coste 110 son seleccionados para generar a dos nuevos individuos para la siguiente generación, el proceso de aplicación es el del cruzamiento uniforme ordenado que se había ilustrado anteriormente, por lo que se mostrarán los resultados en una tabla análoga que ilustra la generación de los dos nuevos individuos.

A	D	C	E	B	Individuo 1
A	B	D	C	E	Individuo 2
1	0	1	0	1	Máscara
A		C		B	Hijo 1 (se copia del ind 1, faltan D y E)
A		D		E	Hijo 2 (se copia del ind 2, faltan B y C)
A	D	C	E	B	Individuo 1
A	B	D	C	E	Individuo 2
1	0	1	0	1	Máscara
A	D	C	E	B	Hijo 1 (D aparece antes que E en ind 2)
A	C	D	B	E	Hijo 2 (C aparece antes que B en ind 1)

Figura 4.12 Los dos pasos para generar un par de descendientes de una pareja de individuos con cruzamiento uniforme ordenado.

A partir de los cromosomas ADCEB y ABDCE se han generado los dos nuevos individuos ADCEB y ACDBE, uno de ellos duplica a uno de los padres mientras que el otro es nuevo con coste de 130 y pocas probabilidades de ser elegido en una nueva generación.

Por último se muestra un esquema de mutación que permita ejemplificar la aplicación de este operador genético en el caso de estudio.

Se selecciono para aplicar el operador de mutación a él cromosoma ACDEB con coste de 100, la siguiente tabla ilustra el proceso de la mutación explicado anteriormente:

A	C	D	E	B	Individuo 1
1	0	1	0	1	M aleatoria
A		D		B	Temp.
A	E	D	C	B	Nuevo individuo

Figura 4.13 Mutación.

El nuevo individuo generado es AEDCB y tiene un coste de 100.

Por supuesto en un ejemplo limitado como éste los avances no son espectaculares en la búsqueda de la solución o incluso pueden parecer un paso atrás, sin embargo el paso de las generaciones es la clave para la superación de la población y al igual que en la naturaleza el tiempo y la selección deberán producir una población más cercana al óptimo.

Si bien la realización de los cálculos a mano es un procedimiento largo y lento que conduce a un proceso de iteración lento para el algoritmo genético, la clave será permitir que sea la computadora quien realice todo el trabajo de cálculo y computo, y éste será precisamente el tema del siguiente capítulo.

CAPÍTULO V

DESARROLLO DEL ALGORITMO

En el anterior capítulo se estudiaron las técnicas que permitirán desarrollar el modelo del algoritmo genético, ahora es tiempo de ocuparse de la automatización de dicho modelo por medio de la programación de una aplicación en computadora que permitirá cumplir con dos objetivos. Por un lado la construcción del modelo del algoritmo genético en un programa para aplicarlo de manera tangible y por otro la comparación con un segundo método de solución del Problema del Agente Viajero, en este caso el algoritmo del mejor vecino. Por lo tanto este capítulo estará dividido en dos secciones, la primera muestra de manera breve la programación del algoritmo del mejor vecino en su forma más simple, y la segunda se ocupará de la programación del algoritmo genético. En ambos casos los diagramas de flujo y el pseudocódigo serán las herramientas auxiliares para explicar mejor este proceso.

En todo momento se usará como guía lo expuesto en los capítulos anteriores, ya que el proceso de codificación tan solo consiste en plasmar en líneas de código lo anteriormente expuesto.

5.1 Lenguaje elegido

El lenguaje elegido para este proyecto es el Visual Basic.NET correspondiente a la plataforma de desarrollo de Microsoft Visual Studio 2003, en su versión de Visual Studio .NET Professional Edition. Por lo tanto al ser ésta una aplicación de escritorio el ambiente de desarrollo será el de Windows Forms, contando como apoyo con el Framework 2.0 de Microsoft. La elección de este lenguaje se basó en la compresión fácil de la codificación de éste y a la mayor robustez que adquirió con respecto a la versión Visual Basic 6.0, ya que .NET ofrece la posibilidad de programación orientada a objetos; lo que permitirá la creación de una clase individuo la cual realizará las operaciones necesarias para la implementación del algoritmo genético; además de permitir ligar los conocimientos de Informática con los de Matemáticas. Por lo demás la facilidad de desarrollar en un ambiente visual para la interfaz de usuario y crear pantallas de captura de datos más accesibles para introducir los datos de la matriz de adyacencia o las coordenadas de las ciudades que integran el problema; es la razón de la elección del lenguaje y plataforma de desarrollo.

5.2 Algoritmo del mejor vecino Diagramas e implementación

Como se expuso anteriormente el algoritmo del mejor vecino, consiste tan solo en unir la ciudad inicial con la más cercana y continuar con este procedimiento hasta que no queden ciudades por visitar, en ese momento se realiza el recorrido final hacia la ciudad inicial y se da por concluido el problema, obteniendo una solución válida. Anteriormente se observo que el problema consiste en realizar un circuito Hamiltoniano usando el vértice más cercano y también que el algoritmo de solución

fue enunciado en 1977 por Rosenkrantz, Stearns y Lewis. Siguiendo dicho algoritmo como guía se tiene que:

```
Inicio
        t, j vértices
        V(j) arreglo de vértices a visitar
        Seleccione un vértice j al azar
        t = j
        Eliminar j del arreglo V(j)
        Mientras el arreglo V(j) tenga elementos
                Seleccionar de V(j) el vértice j tal que la distancia de t a j sea
                mínima
                Conectar t a j
                t = j
                Eliminar j del arreglo V(j)
        Fin mientras
Fin
```

De manera que se manejara un arreglo con las ciudades para poder comparar los costos de los viajes entre éstas. Dicho arreglo será una matriz de incidencia a la que se llamará *MatrizIncidenciaMV* como referencia a que es la matriz de incidencia del Mejor Vecino. Teniéndose para el diagrama 5.1 la matriz de la figura 5.2

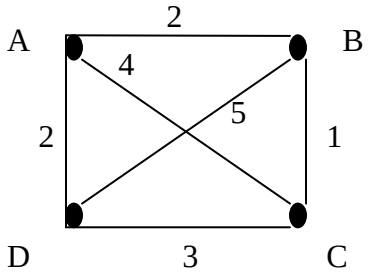


Figura 5.1

	A	B	C	D
A	∞	2	4	2
B	2	∞	1	5
C	4	1	∞	3
D	2	5	3	∞

Figura 5.2 *MatrizIncidenciaMV*

El recorrido iniciará en A y terminará en A. En este caso como en muchos otros existen dos opciones con el mismo coste ya que viajar desde A hacia B tiene un coste de 2 igualmente que ir de A hacia D, en éste caso el algoritmo desempata arbitrariamente, él programa utilizará como criterio el orden alfabético, de esta manera el recorrido quedaría A-B-C-D-A. Se puede apreciar que al iniciar el algoritmo se encuentra ubicado en el primer renglón de la matriz, y de ahí decide a

cual ir analizando los costes del renglón y eligiendo el de menor costo, en este caso la columna B. Es de notar que las distancias de la diagonal principal son marcadas con un número muy alto, para evitar que se tome como opción ir de una ciudad hacia si misma. Una vez que se llega a B él punto de decisión es el renglón dos, desde ahí se debe decidir el siguiente paso, descontando como opción a la ciudad B y la ciudad A debido a que estas ciudades ya fueron visitadas. Continuando con este procedimiento se consigue encontrar la solución, se pueden diferenciar dos pasos básicos en este procedimiento.

1. Elegir la ciudad a visitar entre las ciudades no visitadas.
2. Marcar la ciudad como visitada cuando ya fue elegida.

Él algoritmo mejor vecino quedará de la siguiente manera en Diagrama de primer nivel:

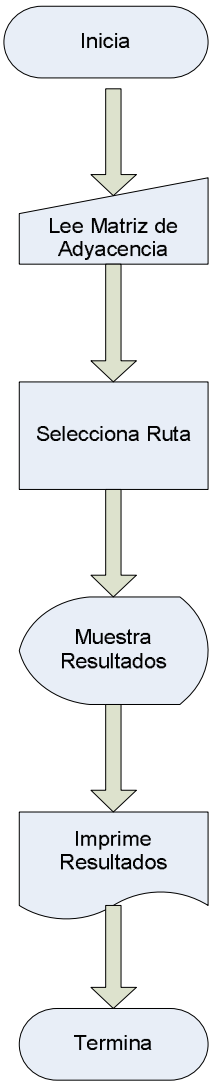


Figura 5.3 Diagrama Nivel 1 algoritmo Mejor Vecino

El primer paso consistente en leer la matriz de adyacencia no es complicado y él programa lo hará por medio de dos métodos. Realizando el llenado de la matriz directamente en una pantalla de captura o indicando las coordenadas de cada punto,

en cuyo caso el programa calculará los valores por medio de la fórmula de la distancia entre dos puntos en el plano y con esta información se llenará la matriz de incidencia. Este paso resulta trivial de programar y no será explicado en detalle en su implementación, pero el proceso sí será ejemplificado en el siguiente capítulo.

En cuanto a los pasos *Mostrar Resultados* e *Imprime Resultados*, consistirán en la visualización de la cadena de solución, en este caso A-B-C-D-A y el coste del recorrido que para este ejemplo es de 8, primero en la pantalla y después guardándolo en un archivo de texto para su consulta y posterior impresión. Ambos pasos serán mostrados en el siguiente capítulo y su codificación será dejada de lado por resultar relativamente sencilla. Concentrando de este modo la explicación en el paso de obtener la ruta solución del problema; de acuerdo a lo anteriormente expuesto se tienen los siguientes pasos en nivel 2, ilustrados por la figura 5.4

Se observa que los recorridos inician en A por que todo recorrido iniciará en esa letra, el costo es cero al iniciar, a continuación se pregunta si todas las ciudades están marcadas como visitadas en cuyo caso se cierra el recorrido viajando hacia A y se suma al costo el viaje de regreso hacía A desde el último nodo visitado. En caso contrario se debe decidir cual es la siguiente ciudad a visitar, sumar este recorrido a el costo y marcar la ciudad como visitada. Esto plantea dos cuestiones:

1. ¿Cómo decidir cuál es la siguiente ciudad a visitar?

2. ¿Cómo marcar como visitada una ciudad?

La búsqueda de la ciudad a visitar se realiza por medio del recorrido del renglón, iniciando en el primer renglón ya que ahí inicia todo recorrido y después en el renglón indicado por la anterior ciudad elegida, de este modo bastará con elegir el mínimo valor de esta columna para obtener la siguiente ciudad a visitar.

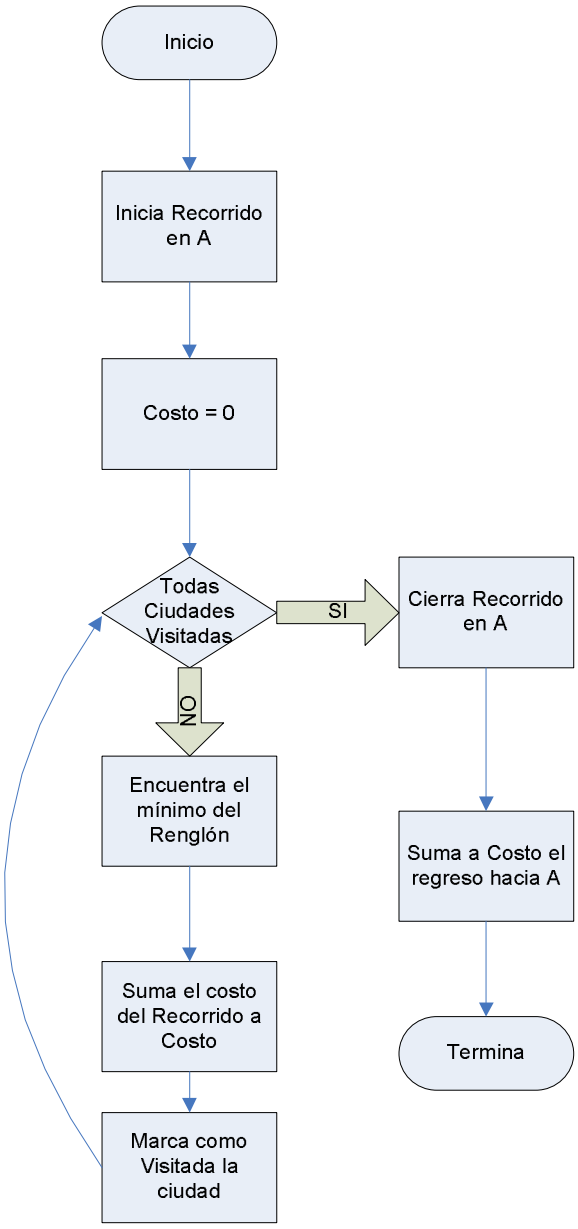


Figura 5.4 Diagrama de datos Nivel 2 Algoritmo Mejor Vecino

Por ejemplo:

	A	B	C	D
A	∞	2	4	2
B	2	∞	1	5
C	4	1	∞	3
D	2	5	3	∞

Figura 5.5 Ejemplo de selección de renglón.

Supónganse las variables *Recorrido* como una cadena de caracteres y *Costo* como un entero positivo.

1. Iniciar en el renglón 1, el recorrido inicia en A y el costo es igual a cero.

Recorrido = “A”
Costo = 0

2. Recorre el renglón uno y rompe el empate por orden alfabético seleccionando la columna dos con el valor 2. Así se tiene que:

Recorrido = “A-B”
Costo = 2

3. Desde el renglón dos se selecciona la columna C con valor de 1 y se tiene:

Recorrido = “A-B-C”
Costo = 3

4. Desde el renglón tres se selecciona el valor mas pequeño ¿Columna B valor 1?

Aquí se presenta el problema mencionado anteriormente en el segundo punto, al situarse en el renglón C y buscar el valor más pequeño se obtiene que el menor es la columna B con valor de 1 pero ya fue visitada Esto plantea la cuestión de ¿Cómo marcar las ciudades como ya visitadas?

La respuesta es el valor simétrico, al seleccionar una ciudad se debe hacer 0 su valor o un valor muy alto para que no se tome en cuenta al momento de seleccionar la siguiente ciudad y lo mismo se debe hacer con su simétrico para evitar los retornos. Adicionalmente también se convierte en cero toda la columna de valores de la ciudad visitada para impedir el retorno a ella, y se hace cero el valor de la columna de A que corresponde a esa ciudad para impedir el retorno hacia A antes de visitar el resto de las ciudades.

Ahora el procedimiento queda como sigue:

1. Inicia en el renglón 1 el recorrido inicia en A y el costo es igual a cero.

Recorrido = “A”
Costo = 0

	A	B	C	D
A	∞	2	4	2
B	2	∞	1	5
C	4	1	∞	3
D	2	5	3	∞

Figura 5.6

2. Se recorre el renglón A y se rompe el empate por orden alfabético seleccionando la columna B con el valor 2. Así se tiene que:

Recorrido = "A-B"
Costo = 2

Se marca la ciudad como visitada con valor cero (Los valores cero no serán tomados en cuenta al buscar el mínimo de renglón).
Se hace 0 el valor (1,2) y también el (2,1), además de la columna B y el valor del renglón B en la columna de A.

	A	B	C	D
A	∞	0	4	2
B	0	∞	1	5
C	4	0	∞	3
D	2	0	3	∞

Figura 5.7

3. Desde el renglón B se selecciona la columna C con valor de 1 y se tiene:

Recorrido = "A-B-C"
Costo = 3
Se marca la ciudad C como visitada con valor cero. Se hace 0 el valor (2,3) y también el (3,2), además de la columna C y el valor del renglón C en la columna de A.

	A	B	C	D
A	∞	0	0	2
B	0	∞	0	5
C	0	0	∞	3
D	2	0	0	∞

Figura 5.8

4. Desde el renglón C se busca el mínimo valor y se encuentra que es la columna D, valor 3

Recorrido = "A-B-C-D"
Costo = 6

Se marca la ciudad D como visitada con valor cero Se hace 0 el valor (3,4) y también el (4,3), además de la columna D y valor del renglón D en la columna de A.

	A	B	C	D
A	∞	0	0	0
B	0	∞	0	0
C	0	0	∞	0
D	0	0	0	∞

Figura 5.9

5. Desde el renglón D se busca el valor mínimo y se encuentra que no existe ninguno, así que es el momento de volver a la ciudad A, el valor puede no ser eliminado sin antes confirmar si es el único en el renglón o simplemente guardar con anterioridad los valores de la columna de A y recuperar el último retorno con el valor del último renglón.

Recorrido = "A-B-C-D-A"
Costo = 8

Se ilustra el proceso por medio de pseudocódigo.

Algoritmo del mejor vecino:

```
Procedimiento MejorVecino ()
   'Ejecuta el algoritmo del mejor vecino

   'Variables
   i Entero
   j Entero
   k Entero
   PosMinimo Entero
   Ciudad Cadena
   RecorridoMV() Arreglo de caracteres
   SumaRecorridoMV Entero
   ColumnaAMV Arreglo de enteros

   'NumCiudades tiene el dato que indica en número de ciudades del
   problema
   NumCiudades Entero

   'Redimensiona el arreglo de caracteres de acuerdo al número de
   ciudades
   ReDim RecorridoMV(NumCiudades + 1)
   'Inicia recorrido en A y el costo en 0
   'Inicia la cadena de Recorrido
   RecorridoMV(0) = "A"
   'Inicia la suma del recorrido
   SumaRecorridoMV = 0

   'Posición inicio
   j = 1
   'Guarda valores de A
   'aquí se salva la columna de valores de A
   ReDim ColumnaAMV(NumCiudades + 1)
```

```
   For k = 1 To NumCiudades
     ColumnaAMV(k) = MatrizIncidenciaMV(k, 1)
   Next k

   'Anula la columna de A
   'aquí se vuelve cero la columna de A para evitar regresos
   For k = 1 To NumCiudades
         MatrizIncidenciaMV(k, j) = 0
   Next k

   For i = 1 To NumCiudades - 1
      'Encuentra el minimo  del renglon
      PosMinimo = PosMinimoRenglon(j)
      'Coloca el siguiente valor en la cadena de recorrido
      Ciudad = SigCiudad(PosMinimo)
      RecorridoMV(i) = Ciudad
      'Suma la ciudad al recorrido
      SumaRecorridoMV += MatrizIncidenciaMV(j, PosMinimo)
      'Coloca un cero en el inverso para anular la ciudad
      MatrizIncidenciaMV(PosMinimo, j) = 0
     'Coloca ceros en esa columna para no volver a esa ciudad
      For k = 1 To NumCiudades
         MatrizIncidenciaMV(k, PosMinimo) = 0
      Next k
      j = PosMinimo
   Next i
   'Aquí termina el ciclo solo resta regresar hacia A
   'Añade el regreso a A
   SumaRecorridoMV += ColumnaAMV(PosMinimo)

   'Imprime los valores
   SumaRecorridoMV 'Costo del viaje
   CreaRecorrido(RecorridoMV) & "A" 'Recorrido añadiendo A al final

Fin del procedimiento
```

De esta forma se tiene que hay dos funciones auxiliares para este algoritmo, estas funciones son:

1. SigCiudad(Mínimo) Esta función sólo realiza la tarea de devolver la ciudad correspondiente al número de columna seleccionado de acuerdo con el mínimo de renglón elegido.
2. PosMinimoRenglon(Renglón) A esta función se le envía el número de renglón del que se desea obtener el mínimo y merece una exposición más detallada de su funcionamiento.

```
Función PosMinimoRenglon(renglón Entero) devuelve un entero
   i, j Enteros
   minimo Entero
   PosMinimo Entero

   j = 1
   i = renglon
   'A mínimo se le asigna un valor muy alto para iniciar
   minimo = 10000000
```

```
   For j = 1 To NumCiudades
      If MatrizIncidenciaMV(i, j) <> 0 Then
         If MatrizIncidenciaMV(i, j) < minimo Then
            PosMinimo = j
            minimo = MatrizIncidenciaMV(i, j)
         End If
      End If
   Next j

   Regresa PosMinimo
 Fin Función
```

Así queda concluida la exposición sobre la implementación del algoritmo del mejor vecino.

5.3 Algoritmo Genético Diagramas e implementación

Estudiando ahora el algoritmo genético y su implementación, se tienen los siguientes pasos básicos de acuerdo al capítulo 2:

1. Generar una población inicial aleatoria de n>0 individuos donde cada uno de ellos representa una solución potencial al problema.

2. Seleccionar a los individuos más aptos de acuerdo a los resultados de evaluarlos con una función de aptitud (fitness).

3. Aplicar operadores de reproducción (cruza, mutación, etc) para asegurar el intercambio genético y la creación de una nueva generación de individuos.

4. Iterar hasta alcanzar la condición de paro.

En esté caso se obtiene el problema consistente en salir de un sitio o ciudad A debiendo visitar todas y cada una de las ciudades restantes sólo una vez y entonces retornar hacia A. Por lo tanto se tendrá un espacio de búsqueda que consistirá en las permutaciones de n sitios, cada permutación de los n sitios produce una solución válida (que es un viaje completo de n sitios) la solución óptima es una permutación que produce el mínimo costo del viaje. El tamaño del espacio de búsqueda es n!.

Así se tiene que:

10 sitios: 3 628 800 posibles rutas a seguir.
20 sitios: 2.43×10^{18}
30 sitios: 2.65×10^{32}

La puntuación o fitness dará el costo total del recorrido, es decir la suma de los viajes para ir desde A hasta visitar todas las ciudades una vez y volver hacia A.

El criterio a optimizar por supuesto consistirá en encontrar la ruta que minimice el costo del recorrido.

La información de entrada será la distancia individual que existe entre cada uno de los puntos.

La codificación de la solución del problema será la natural ruta de ciudades, donde A-B-C-D indica el orden en que deben ser recorridas las ciudades.

La selección se realiza mediante el método de ruleta, mejorado mediante "*elitismo*", que conserva los 5 mejores cromosomas de una generación para la siguiente, evitando que se pierda este valioso material genético.

El cruce se construye seleccionando una subsecuencia de un viaje, escogiendo dos puntos de corte, que sirven como límites para las operaciones de intercambio.

La siguiente figura ilustra esté concepto; desde luego que la combinación de dos padres por medio del cruce puede dar como resultado un individuo erróneo o equivocado, donde se repitan ciudades, queden ciudades sin visitar o se de el caso en que el recorrido no inicie o no termine en A. Para evitar este problema se recurrirá a un proceso auxiliar de corrección de la cadena que permitirá convertir un individuo inviable en uno válido alterando lo menos posible la herencia de material genético de los padres hacia él hijo.

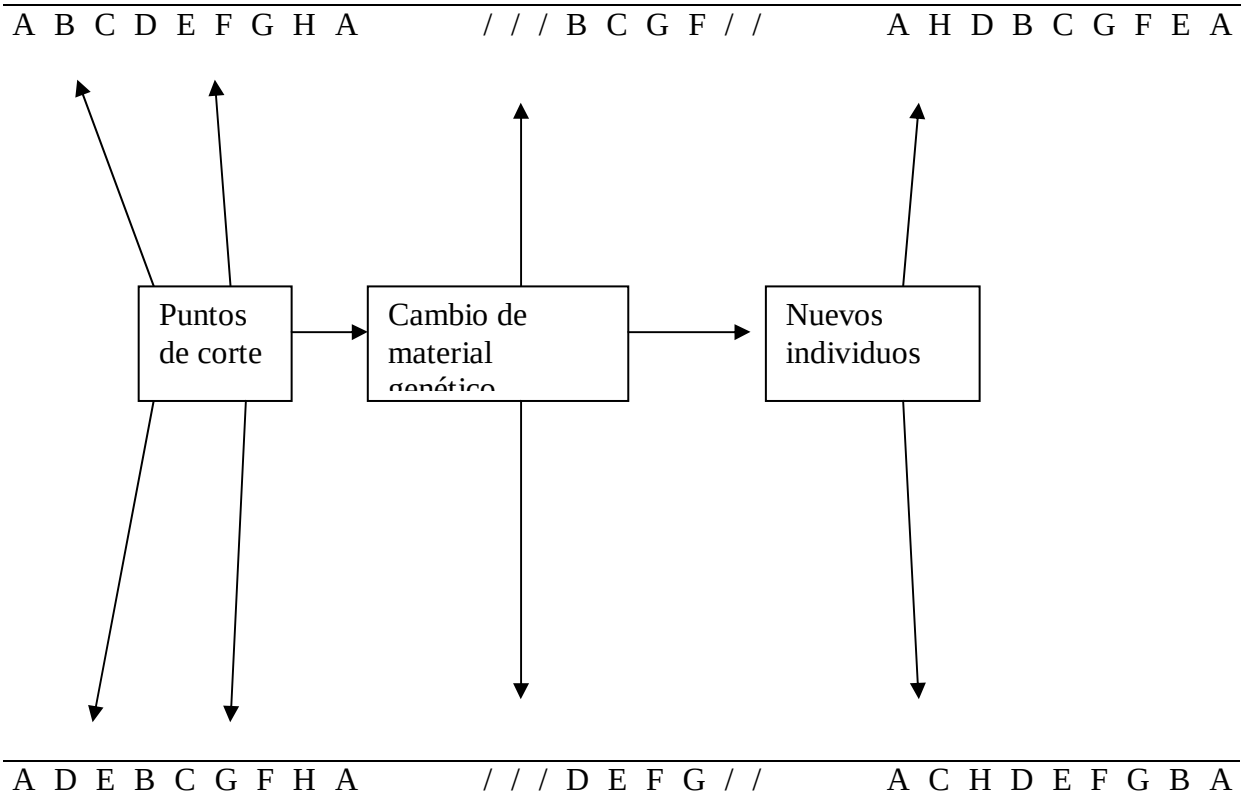


Figura 5.10

En cuanto a la mutación se tiene que el operador intercambiará dos posiciones dentro del cromosoma, así:

A – B – C – D – E – F – G – A

Puede cambiar a:

A – B – **G** – D – E – F – **C** – A

En el caso de la mutación la única situación en que podríamos generar por error un individuo no viable, consistiría en que al realizar el intercambio de posiciones resultara seleccionada la ciudad donde inicia y termina el recorrido, en este caso la ciudad A y de ocurrir esto se cambiara por otra.

Ejemplo:

A – B – C – D – E – F – A

Se cambia por:

A – B – C – D – A – F – E
Lo cual es un error, por lo que la elección del miembro a intercambiar se limitará a las posiciones internas dentro del cromosoma, dejando fuera de este proceso a la ciudad inicial y final del recorrido.

Así se tiene el diagrama de flujo de datos en nivel 0, mostrado en la figura 5.11, donde se observan como los pasos iniciales crear la primer generación y evaluarla, esto permite tomarla como grupo inicial de creación para la siguiente generación, la creación de esta primer generación no conlleva el uso de operadores genéticos sino que es creada de modo aleatorio, por lo que no se esperan individuos con buenas puntuaciones de solución.

El paso de evaluación será igual en todas las generaciones y consiste en obtener el costo del recorrido del individuo.

Por último el criterio de paro del algoritmo, llamado en el diagrama "*termina tiempo*" consistirá en alcanzar el número de generaciones estipulado por los parámetros al inicio del algoritmo.

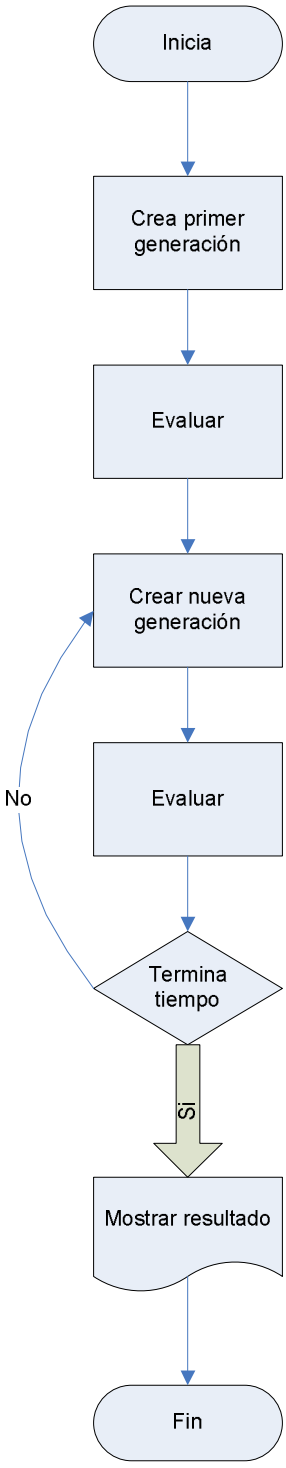


Figura 5.11

En cuanto a la generación de cada generación se tiene el diagrama 5.12, del cual se debe recordar que solo aplicará a partir de la segunda generación.

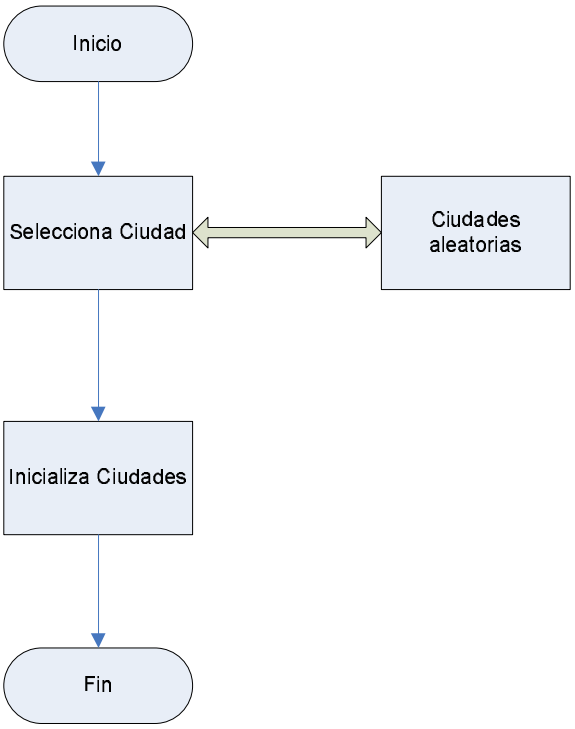


Figura 5.12 Creación de generación

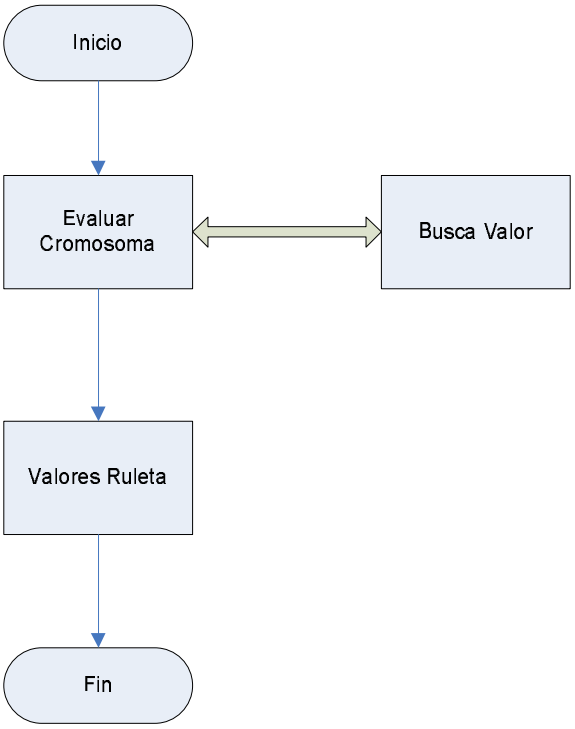


Figura 5.13 Selección

La evaluación para la selección se realiza de acuerdo al diagrama 5.8, en la parte de buscar valor se calcula el costo de la ruta; así se asigna el valor de la ruleta que

servirá para elegir a los individuos para la siguiente generación. Dicho proceso se ilustra en el siguiente diagrama:

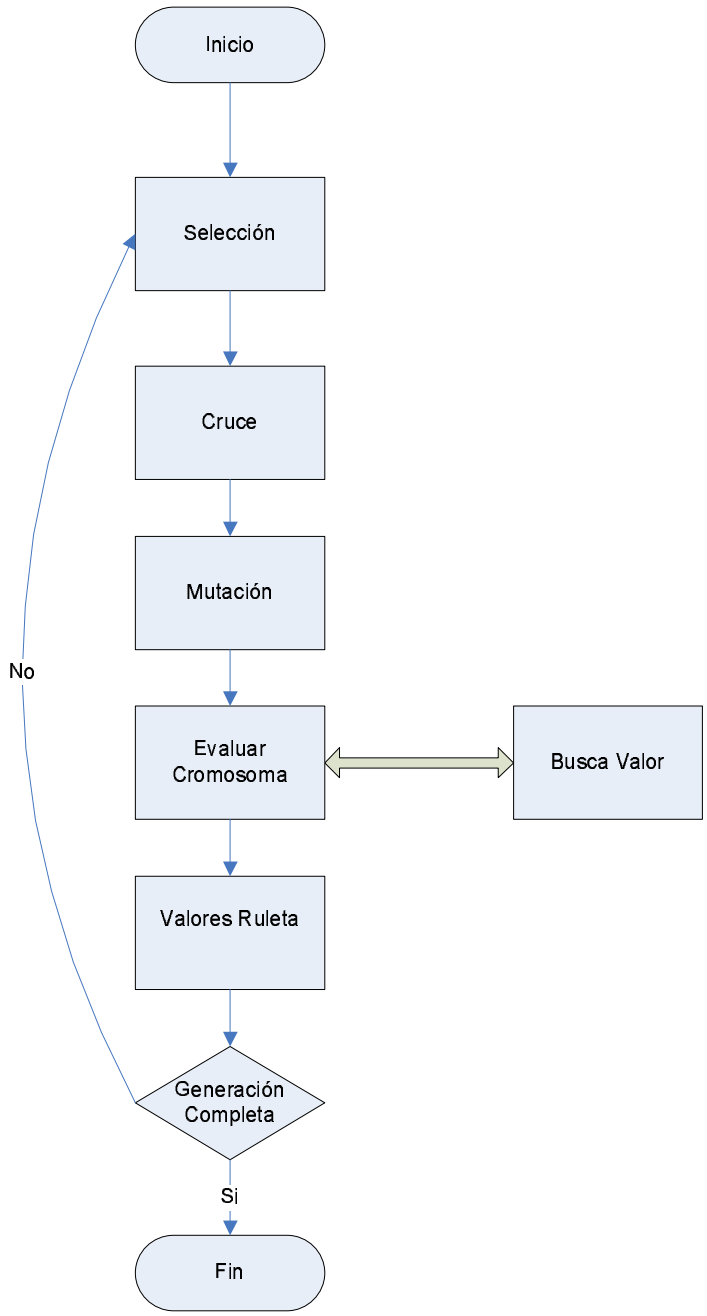


Figura 5.14

Ahora se procede a describir con pseudocódigo los procesos antes descritos. Es de hacer notar que aplicando la programación orientada a objetos se desarrolla una clase individuo con el siguiente diseño que se expone a continuación mediante el uso del lenguaje unificado de modelado (UML):

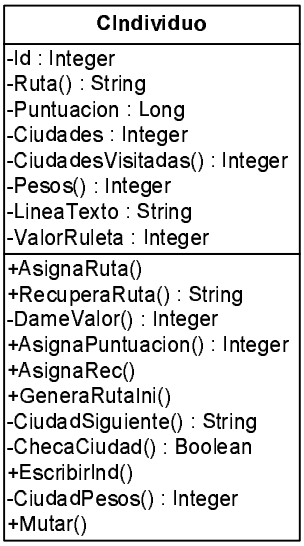


Figura 5.15 Clase Individuo

En los atributos de esta clase están los siguientes con sus respectivas descripciones como comentario:

```
Privado m_Id Entero 'Id individuo
Privado m_Ruta(25) Cadena 'Ruta de recorrido
Privado m_Puntuacíon Entero largo 'Cantidad del recorrido
Privado m_Ciudades Entero 'Número de Ciudades
Privado CiudadesVisitadas(24) Arreglo de enteros
Privado m_Pesos(25) Entero 'Ciudades del recorrido
Privado m_LineaTexto Cadena 'Línea con la información
Privado m_valorRuleta Entero 'Valor para selección en ruleta
```

De acuerdo con los esquemas anteriores se tiene el siguiente pseudocódigo para la ejecución del algoritmo genético.

```
'Variables globales
Publica Pueblo(200) CIndividuo 'Arreglo de 200 individuos máximo
Publica RutaAG(100) Cadena
Publica Poblacion Entero 'Número de Población
Publica NumGeneraciones Entero 'Número de generaciones
Publica ProbMutacion Entero 'Probabilidad de mutación
Publica DistTotalAG Entero Largo 'Distancia Total
Publica DistPromAG Entero Largo  'Distancia Promedio
Publica PuebloAux(200) CIndividuo 'Población auxiliar
Publica ProbMuta As Long 'Probabilidad de mutación




Private Genetico ()
    'Variables locales
```

```
    Dim icount Entero
    Dim icounts Entero
    Dim icontador Entero
    Dim indicepuebloaux Entero
    Dim TiempoInicio Fecha
    Dim MejorSolucion Entero Largo
    Dim idMejorSolucion Entero
    Dim RecorridoFinal Cadena

    'Inica el conteo de ejecución del algoritmo
    TiempoInicio = Now

    'Aplica el algoritmo genetico
    ReDim Pueblo(Poblacion) 'Redimensiona la población

    'Genera población inicial
    'Inicializa la matrices de la población
    For icount = 0 To Poblacion
        Pueblo(icount) = Nuevo CIndividuo
    Next icount

    'Inicializa una población auxiliar
    ReDim PuebloAux(Poblacion)
    For icounts = 0 To Poblacion
        PuebloAux(icounts) = Nuevo CIndividuo
    Next icounts

    For icount = 0 To Poblacion
        PuebloAux(icount).GeneraRutaIni(icount) 'Crea Generación
    Next icount

    'Crea la primer generación
    For icount = 0 To Poblacion
        Pueblo(icount).GeneraRutaIni(icount) 'Crea Generación
    Next icount

    'Asigna valores de ruleta
    IdConsecutivo = Población

    'Guarda datos en archivo
    ArchivoGeneracion()

    'itera hasta llegar al número de generaciones deseado
    For icount = 1 To NumGeneraciones

        'Selecciona índividuos por elitismo para siguiente
generación
        Elitismo()

        'Asigna valores de ruleta
        RuletaVal()

        'Selecciona por cruce y crea los nuevos individuos
        indicepuebloaux = 5

        For icounts = 0 To Poblacion - 5
            Cruce(indicepuebloaux)
            indicepuebloaux += 1
```

```
            Next icounts

            'Pasa población auxiliar a la actual
            For icontador = 0 To Poblacion
                Pueblo(icontador) = PuebloAux(icontador).Clone
            Next icontador

            'Mutación
            Mutar()

            'Anexa a archivo de generaciones
            ArchivoGeneracionSiguiente(CStr(icount + 1))

        Next icount

        'Elije mejor respuesta
        MejorSolucion = Pueblo(0).Puntuacion
        idMejorSolucion = 0
        For icontador = 0 To Poblacion
            If Pueblo(icount).Puntuacion < MejorSolucion Then
                MejorSolucion = Pueblo(icount).Puntuacion
                idMejorSolucion = icount
            End If
        Next icontador

        'Coloca los resultados

        'Distancia recorrida
        txtDistAG = Pueblo(idMejorSolucion).Puntuacion
        RecorridoFinal = ""

        For icount = 0 To NumCiudades
             RecorridoFinal = RecorridoFinal & _
             Pueblo(idMejorSolucion).RecuperaRuta(icount)
            If icount <> NumCiudades Then
                RecorridoFinal = RecorridoFinal & ", "
            End If
        Next icount

        'Recorrido de solución
        txtRecAG = RecorridoFinal

        'Finaliza el tiempo de ejecución del algoritmo

        'Tiempo total de ejecución del algoritmo
        'Resta la hora actual a la de inicio
        txtTiempoAG = TiempoInicio.Subtract(Now)

    Fin
```

Se analizará ahora el funcionamiento de las funciones más importantes utilizadas durante el proceso.

La función ArchivoGeneración, realiza la escritura de cada generación en un archivo de texto, con la finalidad de permitir dar seguimiento a la evolución de la población de soluciones. Su programación es sencilla, baste con decir que utiliza el atributo de

la clase CIndividuo llamado LineaTexto, donde esta almacenada la siguiente información del individuo.

- Id de identidad.
- Cadena de solución.
- Puntuación.

Esta función no pertenece al grupo de funciones que determinan la funcionalidad de la clase CIndividuo, ya que interacciona con la apertura y cierre de archivos para escritura, se prefirió dejarla fuera del cuerpo de ésta.

La función *GeneraRutaIni* consta de la siguiente definición.

```
Publica Sub GeneraRutaIni(indice Entero)
icount Entero
numAleatorio Entero
newCiudad Cadena
CiudadCorrecta Bolean

 Randomize()'Permite generar números aleatorios distintos
 ReDim CiudadesVisitadas(NumCiudades) 'Redimensiona
 'Marca las ciudades como no visitadas o no existentes
 For icount = 0 To NumCiudades - 1
     CiudadesVisitadas(icount) = -1
 Next icount

 Id = indice
 Puntuacion = 0
 Ciudades = NumCiudades
 'Todo recorrido inicia en A
 AsignaRuta(0, "A")
 'Todo recorrido inicia en 1
 AsignaRec(0, 1)
     'Genera el resto del recorrido
 For icount = 1 To NumCiudades - 1
     CiudadCorrecta = False
     While (CiudadCorrecta = False)
          numAleatorio = CInt(Int((NumCiudades - 1) * Rnd()) + 1)
         'Genera aleatorios entre 1 y el número de ciudades
         CiudadCorrecta = ChecaCiudad(numAleatorio)
     End While
     'Marca la ciudad como visitada
     CiudadesVisitadas(icount) = numAleatorio
     'Obtiene la letra de ciudad
     newCiudad = CiudadSiguiente(numAleatorio)
     'Asigna siguiente ciudad
     AsignaRuta(icount, newCiudad)
     'Asigna valor al recorrido
     AsignaRec(icount, (numAleatorio + 1))
 Next icount
 'Cierra el recorrido
 AsignaRuta(NumCiudades, "A") 'Todo recorrido termina en A
 AsignaRec(NumCiudades, 1) 'Todo recorrido termina en 1
 Puntuacion = AsignaPuntuacion() 'Asigna puntuación al individuo
```

```
 EscribirInd() 'Escribe sus datos en una línea
 End Sub
```

Se pueden observar varias funciones integradas en este pseudocódigo, A continuación se realiza un análisis de ellas.

AsignaRuta() es una función que recibe un índice y un carácter que designan una ciudad, la acción realizada por esta función consiste en asignar el carácter de esa ciudad a la propiedad Ruta en la posición indicada por el índice. Por esto es que al inicio asigna la letra A en la posición 0, ya que todo recorrido inicia en A.

AsignaRec() es una función que recibe un índice y un número de ciudad, su trabajo consiste en asignar el número de esa ciudad en el arreglo Pesos en la posición indicada por el índice, de esta manera se guarda un registro de los número de ciudad visitados, esto será útil cuando se calcule la puntuación del individuo. Al inicio asigna el valor 1, es decir la primera ciudad, en la posición cero.

El ciclo for siguiente tiene como función generar el resto del recorrido, dejando como último paso el cierre del recorrido con el viaje hacia A, y asignando el retorno de este punto hacia A. Durante el proceso de generar el recorrido se utiliza una variable de control designada *CiudadCorrecta*, que es un valor boleano que elimina la posibilidad de que se pueda asignar al recorrido como ciudad siguiente (dado que son generadas aleatoriamente) una ciudad ya visitada.

Dentro de este ciclo while se tiene la función *ChecaCiudad*. La cual verifica si la ciudad que le es pasada como argumento, de acuerdo al número aleatorio generado, es una ciudad que ya fue visitada, su definición es la siguiente:

```
Privada Funcion ChecaCiudad(aleatorio Entero) Bolean
    iCount Entero
    Encontrado Bolean

    Encontrado = Verdadero

    For iCount = 0 To NumCiudades
        If CiudadesVisitadas(iCount) > 0 Then
            If CiudadesVisitadas(iCount) = aleatorio Then
                Encontrado = False
                'Cambia a Falso para indicar que el número ya existe
            End If
        End If
    Next iCount

    Regresado Encontrado

   Fin Funcion
```

Una vez terminado el ciclo while se tiene la seguridad de que se ha encontrado una ciudad no visitada. Con esta nueva ciudad a visitar se realizan los siguientes pasos para culminar la iteración:

- Marca la ciudad como visitada.
- Obtiene la letra de la ciudad para colocarla en la nueva ciudad.
- Asigna la ciudad al recorrido en la posición siguiente.
- Asigna el valor del recorrido hacia la nueva ciudad.

Al concluir las iteraciones en el ciclo for, se tiene el recorrido completo, excepto por el regreso hacia A, esta acción se realiza en los últimos pasos. A continuación se explica la función que calcula la puntuación del individuo, es decir el costo del recorrido que se generó para él. Esto se consigue mediante la función *AsignaPuntuación* cuya definición es la siguiente:

```
Privada Funcion AsignaPuntuacion() Entera

    'Calcula el costo del recorrido
    iCount, jCount, kCount Entera
    Puntuacion Entera

    jCount = Pesos(1)
    kCount = 1
    Puntuacion = 0
    For iCount = 2 To NumCiudades + 1
        Puntuacion = Puntuacion + MatrizIncidencia(kCount, jCount)
        kCount = jCount
        jCount = Pesos(iCount)
    Next iCount

    Regresa Puntuacion

   Fin Funcion
```

En esta función se aprovecha la matriz de incidencia para obtener de ella los costos de cada viaje, utilizando para esto el arreglo Pesos, en donde se encuentran almacenados los números de las ciudades visitadas. Así se tiene que para el recorrido A – B – C – D – A, en pesos se almacenaría 1 – 2 – 3 – 4 – 5, de esta manera se usan los contadores iCount, jCount y kCount, para indicarle a la matriz de adyacencia qué valores se necesitan, utilizando los índices de la matriz, que en este caso serian:

(1, 2) Primer recorrido (hacia B)
(2, 3) Recorrido hacia C
(3, 4) Recorrido hacia D
(4, 1) Retorno hacia A desde D

A continuación la función *EscribirInd* captura en el atributo de clase *lineatexto* la información que podrá ser escrita en el archivo de texto por el programa principal.

```
Privada Sub EscribirInd()
    linea Cadena
    lineaAux Cadena
    iCount Entero
```

```
    linea = ""
    lineaAux = ""
    'Coloca el Id del individuo
    linea = CStr(Id) & "   "

    'Forma la cadena del recorrido de ciudades
    For iCount = 0 To NumCiudades
        If iCount <> NumCiudades Then
            lineaAux = lineaAux & CStr(m_Ruta(iCount)) & ", "
        Else
            lineaAux = lineaAux & CStr(m_Ruta(iCount)) & "   "
        End If
    Next iCount
    linea = linea & lineaAux & "   "
    'Coloca la puntuación
    linea = linea & Puntuacion
    m_LineaTexto = linea
```

Fin Sub

Continuando con el análisis del proceso de ejecución del algoritmo genético se tiene ahora la función *Elitismo*(), cuya labor es preservar los 5 mejores individuos para la siguiente generación evitando de esta manera que se pierda este material genético, cabe aclarar que los individuos elegidos, si bien son heredados a la siguiente generación, no son eliminados de la actual, de manera que son elegibles para ser seleccionados padres de un nuevo individuo y también de esta manera aportar su material genético a la siguiente generación, está vez combinándose con otro individuo.

```
Privada Sub Elitismo()
    'Variablea locales
    MejorIndividuo Nuevo CIndividuo
    IdPuntuacion Entero
    icounts Entero
    MejorPuntuacion Entero
    icount Entero

    'Inicia los mejores con un valor negativo
    For icount = 0 To 4
        MejoresId(icount) = -1
    Next icount

    'selecciona los mejores
    For icounts = 0 To 4
        MejorPuntuacion = 10000000
        For icount = 0 To Poblacion
            If MejorPuntuacion > Pueblo(icount).Puntuacion Then
                If Not EstaMejores(Pueblo(icount).Id) Then
                    IdPuntuacion = Pueblo(icount).Id
                    MejoresId(icounts) = Pueblo(icount).Id
                    MejorPuntuacion = Pueblo(icount).Puntuacion
                    MejorIndividuo = Pueblo(icount).Clone
                End If
            End If
        Next icount
```

```
            'Copia el mejor individuo encontrado a la siguiente
            generación
            PuebloAux(icounts) = MejorIndividuo.Clone
        Next icounts
        icount = 7

Fin Sub
```

Dentro de este proceso se destaca la función *EstaMejores*() cuyo objetivo se explica a continuación.

MejoresId es un arreglo de enteros que está destinado a guardar el identificador del individuo seleccionado para la siguiente generación por la técnica de elitismo, la intención es que el mejor individuo, una vez seleccionado ya no vuelva a serlo para obtener de este modo los 5 mejores individuos. Dentro de este contexto se encuentra la función *EstaMejores*(), que busca de acuerdo al identificador del individuo si éste ya existe en el arreglo *MejoresId*, en cuyo caso lo ignora, de no estar ahí, es seleccionado para la siguiente generación e ingresado en el arreglo *MejoresId* para evitar que vuelva a ser seleccionado. Es de destacar que la forma de obtener la nueva generación consiste en almacenar los nuevos individuos en una Población Auxiliar, la cual una vez completa sustituirá al Pueblo actual.

Una vez se tienen a los 5 mejores individuos preservados por medio del elitismo, es tiempo de completar la nueva generación por medio de la creación de nuevos individuos, aplicando el operador genético de Cruce como se observa a continuación.

```
Privada Sub Cruce(indice entero)
'Variables locales
icount Entero
S Entero Largo
numAleatorio Entero
valoracumulado Entero
padre1 Nuevo CIndividuo
padre2 Nuevo CIndividuo
hijo Nuevo CIndividuo
corte Entero
ciudad Cadena
ciudadAux Cadena
corteaux Entero
CiudadPeso Entero


    'Calcula valor de S
    S = 0
    For icount = 0 To Poblacion
        S = S + Pueblo(icount).ValorRuleta
    Next icount

    'Valor aleatorio de ruleta
    Randomize()

    'Genera aleatorios entre 1 y el valor de S
```

```
'Selecciona 1 padre
numAleatorio = CInt(Int((S) * Rnd()) + 1)
valoracumulado = 0
icount = 0
While valoracumulado < numAleatorio
    valoracumulado = Pueblo(icount).ValorRuleta + valoracumulado
    If valoracumulado >= numAleatorio Then
        padre1 = Pueblo(icount).Clone
        valoracumulado = numAleatorio + 1
    Else
        icount += 1
    End If
End While

'Selecciona 2 padre
Randomize()

'Genera aleatorios entre 1 y el valor de S

numAleatorio = CInt(Int((S) * Rnd()) + 1)
valoracumulado = 0
icount = 0
While valoracumulado < numAleatorio
    valoracumulado = Pueblo(icount).ValorRuleta + valoracumulado
    If valoracumulado >= numAleatorio Then
        padre2 = Pueblo(icount).Clone
        valoracumulado = numAleatorio + 1
    Else
        icount += 1
    End If
End While

'Realiza el cruce entre los padres
corte = Int(NumCiudades / 3) 'Elige el punto de corte
IdConsecutivo += 1
hijo = padre1.Clone 'Genes del padre dominante

'Id nuevo que lo identifica como nuevo individuo
hijo.Id = IdConsecutivo     corteaux = corte

'Realiza la herencia del segundo padre
For icount = 1 To corte
    ciudad = padre2.RecuperaRuta(corteaux)
    hijo.AsignaRuta(corteaux, ciudad)
    corteaux += 1
Next icount

'Recompone la cadena
IniciaCadenaBase()

'Pasa los primeros cromosomas a prueba
ReDim CadenaPrueba(NumCiudades)
For icount = 0 To corte - 1
    CadenaPrueba(icount) = hijo.RecuperaRuta(icount)
Next icount

IndicePrueba = corte – 1
```

```
    'Realiza los cambios necesarios
    For icount = corte To NumCiudades - 1
        If Existe(hijo.RecuperaRuta(icount)) Then
            ciudadAux = ObtieneCiudad()
            hijo.AsignaRuta(icount, ciudadAux)
            CadenaPrueba(icount) = hijo.RecuperaRuta(icount)
        Else
            IndicePrueba += 1
            CadenaPrueba(icount) = hijo.RecuperaRuta(icount)
        End If
    Next icount

    'Asigna los nuevos pesos al recorrido
    For icount = 0 To NumCiudades
        ciudad = hijo.RecuperaRuta(icount)
        CiudadPeso = CiudadPesos(ciudad)
        hijo.AsignaRec(icount, CiudadPeso)
    Next icount

    'Copia el nuevo individuo a la siguiente generación
    hijo.Puntuacion = hijo.AsignaPuntuacion()
    hijo.EscribirInd()
    PuebloAux(indice) = hijo.Clone

   Fin Sub
```

Se estudia ahora con más detenimiento la función Cruce; como ya se explicó anteriormente para realizar la operación de cruce, se requiere seleccionar dos padres, en este caso el método de selección que se utilizó fue el método de selección por ruleta, para aplicar dicho método cada individuo debe tener un atributo de puntuación, que permita medir que tan buen candidato a ser padre es. Se considera como criterio de calificación la puntuación; ya que a menor puntuación es mejor candidato a ser padre, el proceso que se siguió para asignar este valor lo realiza la función RuletaVal, misma que se muestra a continuación.

```
Privada Sub RuletaVal()
    icount Entero
    jcount Entero
    inverso Entero

    'Inicializa una población auxiliar para ordenar y otra para
    invertir
    ReDim PuebloOrd(Poblacion)
    ReDim PuebloInv(Poblacion)

    'Copia la población en el arreglo a ordenar
    For icount = 0 To Poblacion
        PuebloOrd(icount) = Pueblo(icount)
    Next icount

    'Ordena el arreglo
    OrdRap(0, Poblacion)

    'Copia la población ordenada en una inversa
    inverso = Poblacion
```

```
    For icount = 0 To Poblacion
        PuebloInv(icount) = PuebloOrd(inverso)
        inverso -= 1
    Next icount

    'Asigna valores de Ruleta a la población ordenada
    For icount = 0 To Poblacion
        PuebloOrd(icount).ValorRuleta = PuebloInv(icount).Puntuacion
    Next icount

    'Asigna los valores de ruleta a la población real
    For icount = 0 To Poblacion
        For jcount = 0 To Poblacion
            If Pueblo(icount).Id = PuebloOrd(jcount).Id Then
            Pueblo(icount).ValorRuleta
=PuebloOrd(jcount).ValorRuleta
            End If
        Next jcount
    Next icount

   Fin Sub
```

La forma de proceder de esta función es simple, primero se copia la población actual a una población auxiliar para ser ordenada, en este caso la población a ordenar es PuebloOrd, dicha población es ordenada de menor a mayor por medio del método de ordenamiento QuickSort recursivo el cual se efectúa en la función OrdRap cuyo cuerpo no se describe por tratarse de un método bien conocido. A continuación se copia esta población ordenada de menor a mayor a otra población, invirtiéndola para tenerla ordenada de mayor a menor, esta población es la designada PuebloInv esto se hace con el fin de pasar los valores de puntuación de la población PuebloInv al atributo ValorRuleta de la población PuebloOrd y desde ahí asignarlo a la población original Pueblo; de esta manera el individuo que tenga el recorrido más corto recibe la puntuación más alta y el individuo con el recorrido más largo recibe la puntuación más pequeña.

Volviendo ahora con el proceso de Cruce, la selección por el método de ruleta especifica el siguiente procedimiento:

- Calcular S, que será la suma de todos los valores fitness de la población actual.
- Generar un número aleatorio A que se encuentre entre 0 y el valor de S.
- Recorrer los individuos de la población acumulando sus valores de fitness, realizando la siguiente pregunta: ¿Si el valor acumulado es mayor o igual que A? Elegir el individuo.

Estos son los primeros pasos que realiza el algoritmo Cruce, la función Randomize permite calcular números aleatorios que son asignados a la variable numAleatorio y a continuación se selecciona el padre dentro de los ciclos while.

Para realizar el cruce, primero se asigna a el nuevo individuo la misma información del primer padre, de este modo el primer padre, que fue seleccionado primero será el

gen dominante, adicionalmente se le asigna un identificador diferente que lo convierta en un nuevo individuo. La herencia del segundo padre se realiza de la siguiente manera:

- Seleccionar un punto de corte con el resultado de la división entera de número de ciudades entre 3.
- Sustituir cromosomas en el hijo, desde el punto de corte cambiando los del padre 1 por el padre 2, tantos como el punto de corte.
- Recomponer la cadena para evitar individuos incorrectos.

Se ejemplifica este proceso:

Sea la cadena A – B – C – D – E – F – A por lo que el número de ciudades es 6, así que el punto de corte es 6 / 3 = 2.

Recordando que un arreglo inicia en el elemento cero se tiene que:

Padre1: A – B – C – D – E – F – A

Padre2: A – B – F – D – C – E – A

Hijo: A – B – C – D – E – F – A

Punto de corte: 2

De acuerdo a los índices

Índice	0	1	2	3	4	5	6
Hijo	A	B	C	D	E	F	A

Figura 5.16

Así que se sustituirá desde la posición 2, es decir desde el valor de C, ¿Cuántos valores serán sustituidos? Tantos como indica el valor del punto de corte es decir dos posiciones, así que los elegidos son los valores de C y D, por lo que se tiene:

Hijo: A – B – X – X – E – F – A

Donde X indica los valores faltantes, mismos que serán sustituidos por los caracteres que ocupan las mismas posiciones en el Padre 2 es decir por F y D.

De esta manera el hijo queda:

Hijo: A – B – F – D – E – F – A

De inmediato es notorio el problema principal del cruce, al sustituir C y D por F y D se perdió el valor C y se tiene ahora duplicado el valor F. ¿Qué se puede hacer para corregir esto?

Él algoritmo ejecutara ahora un proceso de reconstrucción de la cadena que consiste en los siguientes pasos:

1. Construir una cadena base ordenada alfabéticamente con todas las ciudades.
2. Dividir la cadena del hijo en tres segmentos, la primera y la tercera serán ambas del padre 1 y la segunda del padre 2.
3. Copiar a una cadena definitiva el primer segmento.
4. Construir una cadena de prueba con el primer segmento del padre 1.
5. Preguntar para cada elemento de la segunda cadena si ya existe en la cadena de prueba, de ser así, sustituirla por el primer elemento de la cadena base que no exista en la cadena de prueba.
6. Al pasar cada elemento a la cadena definitiva, darlo de alta en la cadena de prueba.
7. Continuar de este modo hasta terminar con el segmento dos y tres, ignorando el último elemento del segmento tres, éste se pasa a la cadena definitiva sin preguntar. (Este elemento siempre será A)

Ejemplificando lo antes expuesto:

Hijo: A – B – F – D – E – F – A

Segmento 1: A – B
Segmento 2: F – D
Segmento 3 E – F – A

Cadena Base: A – B – C – D – E – F
Cadena Prueba: A – B
Cadena Definitiva: A – B

Inicio de iteración

Se pregunta por el primer elemento del segmento dos, en este caso F, se observa que no está en la cadena de prueba, por lo que se pasa a la cadena definitiva y se da de alta en la cadena de prueba:

Cadena Base: A – B – C – D – E – F
Cadena Prueba: A – B – F
Cadena Definitiva: A – B – F

Se pregunta por el siguiente elemento del segmento dos, en este caso D, se observa que no está en la cadena de prueba, por lo que se pasa a la cadena definitiva y se da de alta en la cadena de prueba:

Cadena Base: A – B – C – D – E – F
Cadena Prueba: A – B – F - D
Cadena Definitiva: A – B – F – D

Ahora al no existir más elementos en el segmento dos, se pregunta por el primer elemento del segmento tres, en este caso E, se observa que no está en la cadena de prueba, por lo que se pasa a la cadena definitiva y se da de alta en la cadena de prueba:

Cadena Base: A – B – C – D – E – F
Cadena Prueba: A – B – F – D – E
Cadena Definitiva: A – B – F – D – E

Se pregunta por el siguiente elemento del segmento tres, es decir F, este elemento existe en la cadena de prueba por lo que se sustituye por uno de la cadena base, ¿Cual? El primero que no exista en la cadena de prueba, en este caso es C, por lo que se coloca C en la cadena definitiva y en la de prueba:

Cadena Base: A – B – C – D – E – F
Cadena Prueba: A – B – F – D – E – C
Cadena Definitiva: A – B – F – D – E – C

Como el siguiente elemento del segmento tres es el último pasa directo a la cadena definitiva:

Cadena Definitiva: A – B – F – D – E – C – A

Se copia esta cadena al Hijo y se tiene resuelto el problema:

Padre1: A – B – C – D – E – F – A

Padre2: A – B – F – D – C – E – A

Hijo: A – B – F – D – E – C – A

Nótese un importante detalle, el proceso de recomposición no solo convirtió un individuo no viable en uno viable, sino que además, este hijo es muy similar a sus dos padres, pero sin ser idéntico a ninguno de los dos, lo que es una excelente implementación del proceso de la naturaleza.

Toda esta funcionalidad la realiza él algoritmo auxiliándose con las funciones:

IniciaCadenaBase: La cual crea la cadena base para las sustituciones.

Existe: Que índica si el elemento ya existe en la cadena de prueba.

ObtieneCiudad: Que revisa la cadena base devolviendo el primero elemento que esté en ella y no exista en la cadena de prueba.

En la última parte el algoritmo reacomoda los valores numéricos para obtener la puntuación de la nueva cadena de recorrido.

Se revisará ahora el proceso del último operador genético, la mutación.

Él programa recibe un parámetro que le indica la probabilidad de que ocurra una mutación, con base en este parámetro el algoritmo recorre la población y obtiene un número aleatorio por cada individuo; si el resultado es menor o igual a la probabilidad de mutación realiza el cambio.

Aquí está la función:

```
Privada Sub Mutar()
icount Entero
numAleatorio Entero
IndividuoAux CIndividuo

    For icount = 0 To Poblacion
        'Valor aleatorio de mutación
        Randomize()
         'Genera aleatorios entre 1 y 100 %
        numAleatorio = CInt(Int((100) * Rnd()) + 1)
         If numAleatorio <= ProbMuta Then
             If icount < 5 then
         IndividuoAux= Pueblo(icount).Clone
             End if
            'Realiza mutación
            Pueblo(icount).Mutar()
            'Permite la mutación de los individuos elite solo si es
            'positiva
           If icount < 5 then
         Pueblo(icount) = IndividuoAux.Clone
             End if

        End If
    Next icount
Fin Sub
```

En esta función se decide que individuo debe mutar, donde *ProbMuta* es el parámetro. Se define a continuación la función de *Mutar* que esta en cada individuo de la clase *CIndividuo*.

```
Publica Función Mutar()
    'Obtiene aleatoriamente los puntos a intercambiar
    numAleatorio1 Entero
    numAleatorio2 Entero
    ciudad1 Carácter
    ciudad2 Carácter
    PesoCiudad1 Entero
    PesoCiudad2 Entero

    Randomize()
    'Genera aleatorios entre 1 y numero de ciudades-1
```

```
    numAleatorio1 = CInt(Int((NumCiudades - 1) * Rnd()) + 1)
    Randomize()
    'Genera aleatorios entre 1 y numero de ciudades-1
    numAleatorio2 = CInt(Int((NumCiudades - 1) * Rnd()) + 1)

    ciudad1 = RecuperaRuta(numAleatorio1)
    ciudad2 = RecuperaRuta(numAleatorio2)

    'Primer cambio
    AsignaRuta(numAleatorio1, ciudad2)
    PesoCiudad1 = CiudadPesos(ciudad2)
    AsignaRec(numAleatorio1, PesoCiudad1)

    'Segundo cambio
    AsignaRuta(numAleatorio2, ciudad1)
    PesoCiudad2 = CiudadPesos(ciudad1)
    AsignaRec(numAleatorio2, PesoCiudad2)

    'Asigna puntuación
    Puntuacion = AsignaPuntuacion()
    EscribirInd()

Fin Función
```

La tarea de esta función es la siguiente:

Dado un cromosoma, de n posiciones, elegir aleatoriamente dos de estas para intercambiarlas y realizar así la mutación; la primer y última posición se excluyen, ya que todo recorrido debe iniciar y terminar en A.

Así por ejemplo, con el siguiente cromosoma elegido para mutar:

A – B – D – F – E – C – A

Se Obtienen los números aleatorios 3 y 5 (en este caso la posición 0 y 6 no son elegibles para intercambiarse, por ser el inicio y fin del recorrido). Al mutar, el cromosoma quedará de la siguiente forma:

Posiciones	0	1	2	3	4	5	6
Original	A	B	C	D	E	F	A
Mutado	A	B	C	F	E	D	A

Figura 5.17 Mutación

Como se puede observar la implementación de la mutación de esta manera es sencilla y tiene la ventaja de que no genera de ningún modo individuos incorrectos.

Al terminar la definición e explicación de los procesos de implementación de las funciones del algoritmo se puede mostrar el funcionamiento del programa con

ejemplos y evaluar el desempeño de ambos algoritmos comparativamente. Estos puntos serán el tema del siguiente capítulo.

CAPÍTULO VI

ANÁLISIS DE RESULTADOS

En el capítulo anterior se estudio la implementación del algoritmo genético y del algoritmo del mejor vecino. Ahora se mostrará cómo utilizar el programa que se diseño y por supuesto comparar el desempeño de ambos algoritmos. Para esto se utiliza un caso de estudio que servirá para ejemplificar el uso del programa sirviendo además como un tutorial de aplicación de los algoritmos y uso del programa. Finalmente se presenta un comparativo de desempeño que permite observar el resultado de ambos algoritmos.

6.1 Caso de Estudio

Se empleará el siguiente caso de estudio, definido por la siguiente matriz de adyacencia.

x,y	A	B	C	D	E	F
A	α	3	93	13	33	9
B	3	α	77	42	21	16
C	93	77	α	36	16	28
D	13	42	36	α	56	7
E	33	21	16	56	α	25
F	9	16	28	7	25	α

Figura 6.1 Matriz de adyacencia a resolver.

Se puede observar que es una matriz simétrica que ejemplifica el problema del agente viajero y del mismo modo servirá para el algoritmo del mejor vecino.

El programa fue desarrollado en el ambiente de Microsoft Visual Studio .NET 2003, utilizando como lenguaje de desarrollo el Visual Basic.NET en el ambiente de desarrollo de aplicaciones para Windows (Windows Forms).

Los sistemas operativos que deseen ejecutar una aplicación de Visual Basic .NET deberán tener instalado .NET Framework 2.0. Esté se incluye en el CD adjunto con está tesis misma que contiene el programa del algoritmo genético; el Framework se incluye por medio del archivo Dotnetfx.exe y para su instalación se deben tomar en cuenta los siguientes requisitos:

Para instalar Dotnetfx.exe, se debe tener uno de los siguientes sistemas operativos con Microsoft Internet Explorer 5.01 o posterior instalado en el equipo:

Microsoft® Windows® 98
Microsoft® Windows® 98 Segunda edición

Microsoft® Windows® Millennium Edition (Windows Me)
Microsoft® Windows NT® 4 (Workstation o Server) con Service Pack 6a
Microsoft® Windows® 2000 (Professional, Server, o Advanced Server) con el último Windows service pack y actualizaciones críticas que se pueden obtener en el sitio Web Microsoft Security (www.microsoft.com/security).
Microsoft® Windows® XP (Home o Professional)
Familia de productos de Microsoft® Windows® Server 2003

A continuación se incluyen una serie de recomendaciones realizadas por Microsoft sobre la instalación de Framework y los requerimientos mínimos para la ejecución de aplicaciones .Net, es de señalar que el algoritmo genético es una aplicación de escritorio y no una aplicación de ambiente cliente – servidor, por lo que la mayoría de estas recomendaciones no son necesarias, pero se ha decidido incluirlas como referencia para cualquier eventualidad que se pudiera presentar en diversos escenarios de instalación.

Si se instala Dotnefx.exe en Windows Server 2003 Beta 3, se interrumpirá la versión de .NET Framework que esté instalada con el sistema operativo. Windows Server 2003 Beta 3 instala la versión 1.0.3215 de .NET Framework. Si se instala una versión posterior de .NET Framework, la versión 1.0.3215 será interrumpida. Si se instala una versión posterior, se puede ejecutar y utilizar la versión posterior. Sin embargo, no se podrá utilizar la versión 1.0.3215, ni siquiera tras desinstalar la versión posterior.

Requisitos mínimos de hardware

Escenario CPU RAM necesaria
Cliente Pentium a 90 MHz* 32 MB**
Servidor Pentium a 133 MHz* 128 MB**

*O la CPU mínima requerida para ejecutar el sistema operativo, lo que sea mayor.

**O la RAM mínima necesaria para ejecutar el sistema operativo, lo que sea mayor.

Hardware recomendado

Escenario CPU RAM recomendada
Cliente Pentium a 90 MHz o más rápido 96 MB o más
Servidor Pentium a 133 MHz o más rápido 256 MB o más

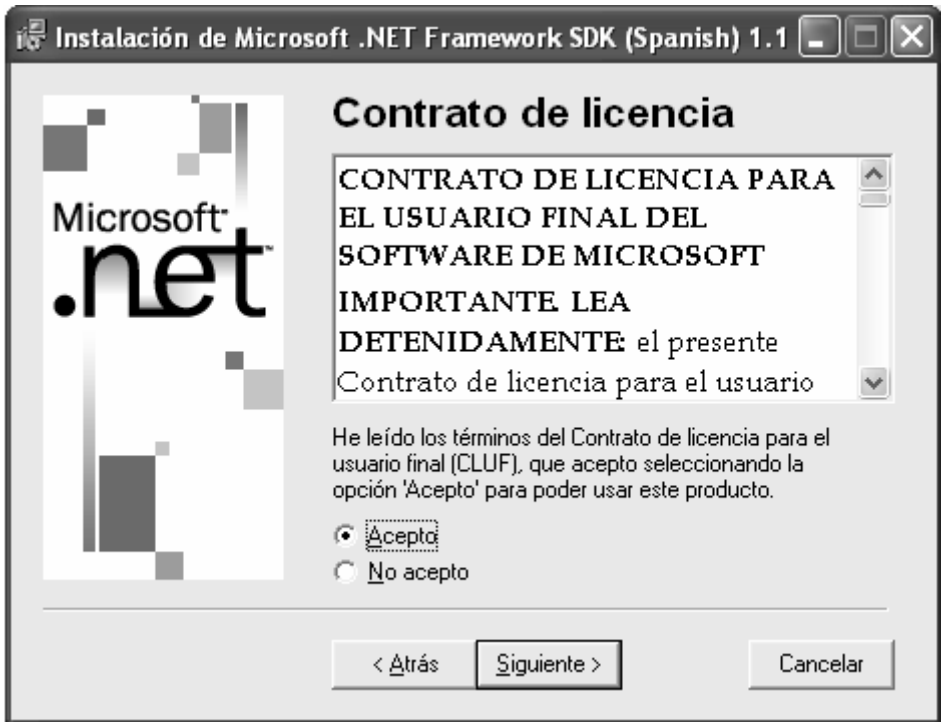
La instalación se realiza a partir del archivo Setup, contenido en la carpeta Release del CD incluido con este trabajo, se recuerda que si no cuenta con Framework se deberá instalar previamente con el archivo Dotnetfx.exe incluido en el CD en la carpeta Framework. Y posteriormente el Framework SDK que se instala con el archivo Setup de la carpeta Framework.

Ejecutando Dotnetfx.exe se tiene:

Figura 6.2 Pantallas de instalación de Framework

Instalando ahora el Framework SDK por medio del archivo Setup que está en la carpeta de Framework.

Las pantallas de instalación son las siguientes:


Instalación de Microsoft .NET Framework SDK (Spanish) 1.1
Contrato de licencia
CONTRATO DE LICENCIA PARA EL USUARIO FINAL DEL SOFTWARE DE MICROSOFT
IMPORTANTE LEA DETENIDAMENTE: el presente Contrato de licencia para el usuario
He leído los términos del Contrato de licencia para el usuario final (CLUF), que acepto seleccionando la opción 'Acepto' para poder usar este producto.
Acepto
No acepto
< Atrás
Siguiente >
Cancelar

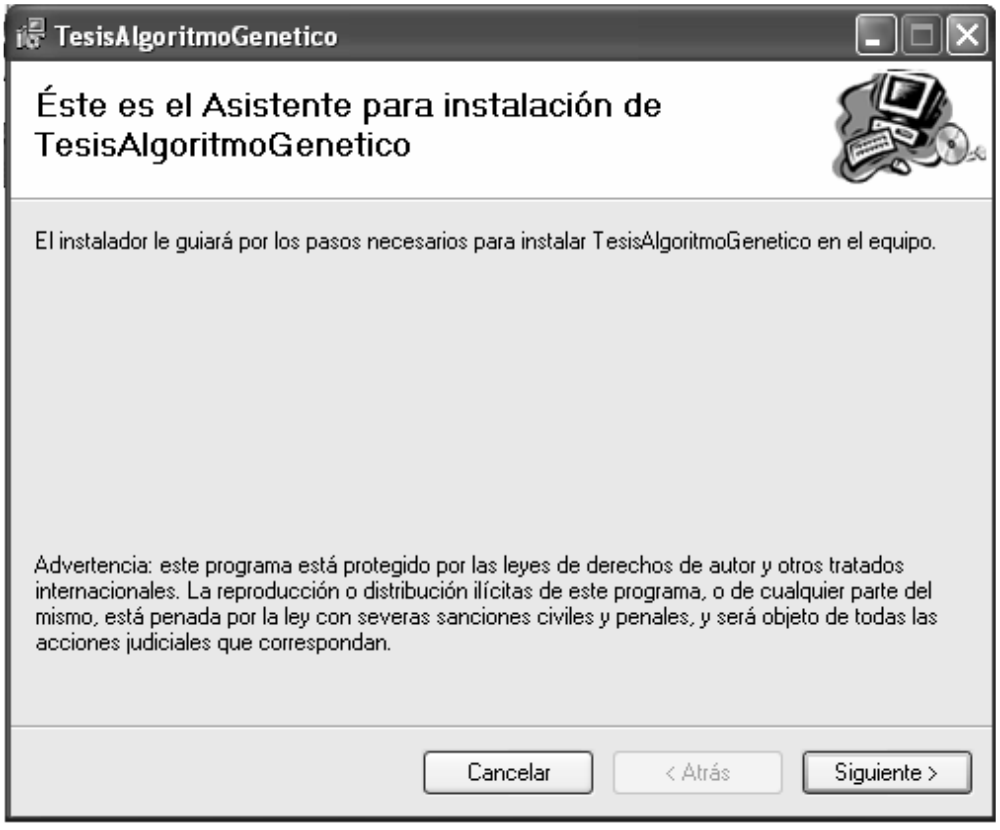

Instalación de Microsoft .NET Framework SDK (Spanish) 1.1
Opciones de instalación
Kit de desarrollo de software
Incluye componentes y herramientas necesarios para generar y ejecutar aplicaciones .NET.
Ejemplos del SDK
Incluye aplicaciones de ejemplo de .NET Framework.
Documentación
Incluye información de desarrollo para .NET Framework SDK.
< Atrás
Siguiente >
Cancelar


Instalación de Microsoft .NET Framework SDK (Spanish) 1.1
Carpeta de destino
Escriba la ruta de acceso de la carpeta en la que desee instalar este producto.
Si el programa de instalación no puede encontrar la ubicación especificada, creará la carpeta necesaria.
Instalar en la carpeta:
C:\Archivos de programa\Microsoft.NET\SDK\
Examinar...
< Atrás
Siguiente >
Cancelar


Instalación de Microsoft .NET Framework SDK (Spanish) 1.1
Instalación de componentes
Los siguientes elementos se están instalando en su PC:
Documentación del producto
Kit de desarrollo de software (SDK)
Ejemplos del SDK
< Atrás
Siguiente >
Cancelar

Figura 6.3 Pantallas de instalación de Framework SDK

Ahora se instala el programa ejecutando el archivo Setup de la carpeta Release. Las pantallas de la instalación del programa del algoritmo genético son ilustradas en las siguientes imágenes.




TesisAlgoritmoGenetico
Seleccionar carpeta de instalación
El instalador instalará TesisAlgoritmoGenetico en la siguiente carpeta.
Para instalarlo en esta carpeta haga clic en "Siguiente". Para instalarlo en una carpeta distinta haga clic en "Examinar".
Carpeta:
C:\Archivos de programa\FES Acatlan\TesisAlgoritmoGenetico\
Examinar...
Espacio en disco...
Instalar TesisAlgoritmoGenetico sólo para este usuario o para todos los usuarios de este equipo:
Para todos los usuarios
Sólo para este usuario
Cancelar
< Atrás
Siguiente >


TesisAlgoritmoGenetico
Confirmar instalación
El instalador está listo para instalar TesisAlgoritmoGenetico en el equipo.
Haga clic en "Siguiente" para iniciar la instalación.
Cancelar
< Atrás
Siguiente >

Figura 6.4 Pantallas de instalación del Programa Algoritmo Genético.

Una vez terminada la instalación se puede ejecutar el programa desde el menú de inicio de Windows, como lo muestra la imagen de la siguiente página.

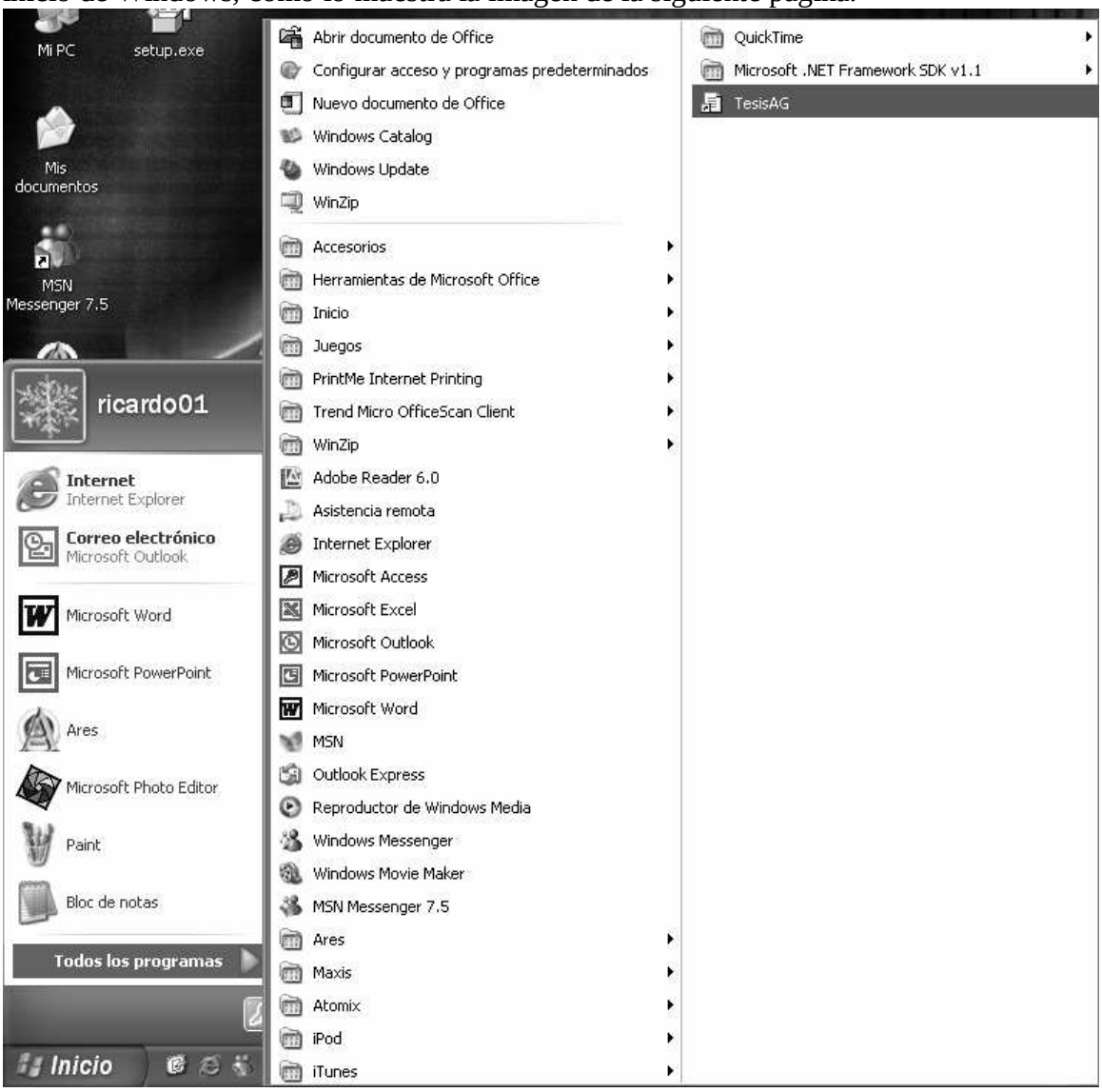


Figura 6.5 Pantalla de inicio del Programa Algoritmo Genético.

La pantalla principal de la aplicación es la siguiente:

Figura 6.6 Pantalla Principal

Los principales parámetros a configurar aparecen del lado superior derecho, junto con el intervalo que deben respetar, los cuales son:

- Número de Ciudades.
- Tamaño de Población.
- Número de Generaciones.
- Probabilidad de Mutación.

El número de ciudades va de un mínimo de tres a un máximo de 25, el tamaño de la población de individuos va desde un mínimo de 50 individuos hasta un máximo de 200, las generaciones desde uno a 50 esto será útil para ver como cambia el comportamiento del algoritmo entre mas generaciones se crean; por último se tiene la probabilidad de mutación, la cual al ser un parámetro configurable permitirá ver la aportación del factor de mutación en la evolución de la solución, este número va de 1 a 100.

A la izquierda se encuentran los botones que permiten ingresar los puntos, ejecutar el algoritmo genético y ejecutar el algoritmo del mejor vecino; del mismo modo en la parte inferior se encuentra el botón de salir para abandonar la aplicación.

Figura 6.7 Botones de Comando.

Los comandos de Algoritmo Genético y Algoritmo Mejor Vecino están inhibidos hasta que se capturen las ciudades mediante el botón **Ingresar Puntos**. Al pulsar este botón se obtiene la siguiente pantalla:

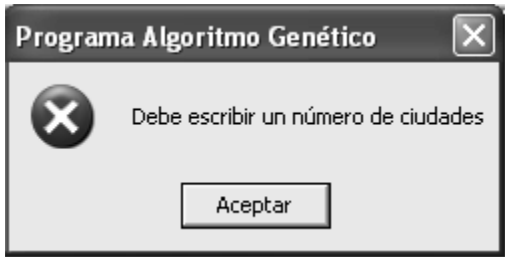


Figura 6.8 Confirmación de número de ciudades.

Este mensaje de validación advierte que primero se deben capturar los parámetros del problema; al capturar y dar clic en el botón se obtiene la siguiente ventana de validación para el tamaño de población:

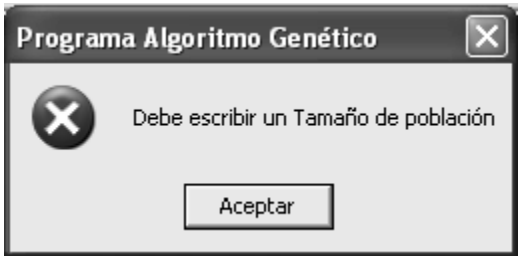


Figura 6.9 Confirmación de tamaño de población.

También se cuenta con validaciones para el número de generaciones y la probabilidad de mutación.

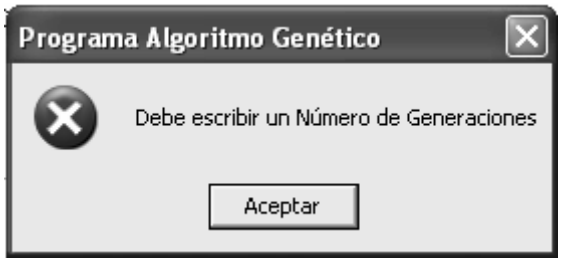


Figura 6.10 Confirmación de número de generaciones

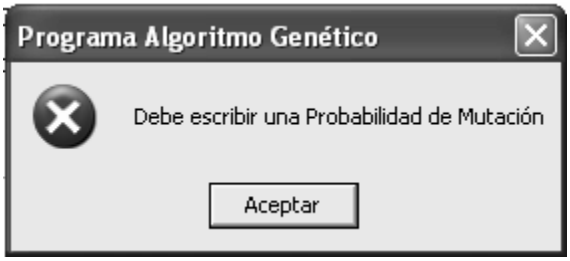


Figura 6.11 Confirmación de probabilidad de mutación

Así mismo se cuenta con las validaciones de valor numérico y del rango de los valores introducidos.

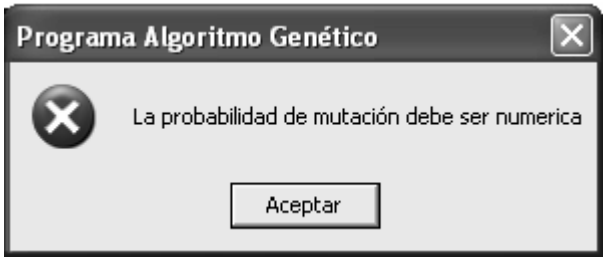


Figura 6.12 Validación de valor numérico en la probabilidad de mutación

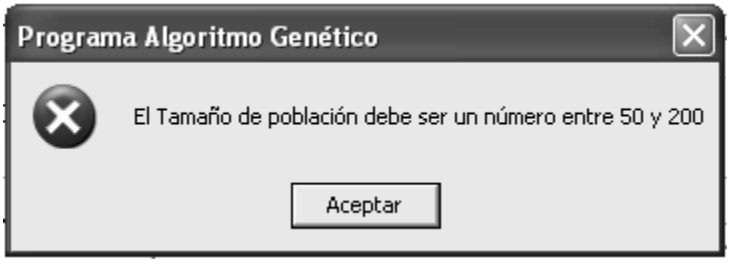


Figura 6.13 Validación de rango en el tamaño de población

Una vez que se capturan estos datos se puede realizar el clic en el botón **Ingresar Puntos** y esto conduce a la siguiente pantalla.

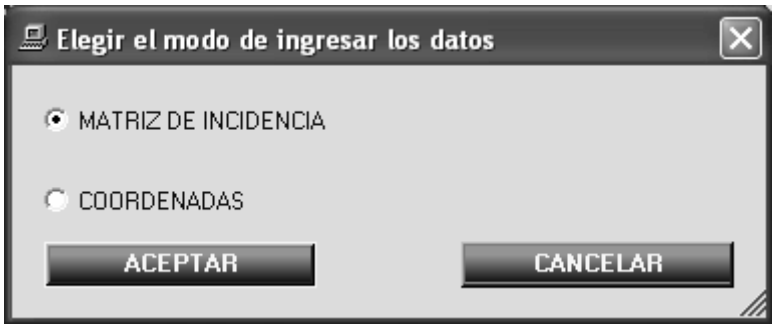


Figura 6.14 Pantalla de opciones para captura de datos.

Esta pantalla permite elegir cual es el medio que se desea utilizar para realizar la captura de los datos del problema, al ingresar por primera vez se encuentra seleccionada la opción de *Matriz de incidencia*, la otra opción es *Coordenadas*, el botón cancelar lleva de regreso a la pantalla anterior. Se selecciona primeramente la opción *Coordenadas* y se obtiene la pantalla mostrada en la figura 6.10, la cantidad de casillas que se encuentran habilitadas depende del número de ciudades que se hayan indicado en la pantalla principal del programa. En él ejemplo de la figura 6.15 se trato del mínimo, es decir tres ciudades; esta pantalla está diseñada para realizar la captura de las coordenadas que definen la posición de un punto en el plano, con esta información y utilizando la fórmula que calcula la distancia entre dos puntos en el plano, se obtiene la información que permite el llenado de la matriz de incidencia, suponiendo que se puede ir desde cada punto hacia todos y cada uno de los restantes puntos, de manera que todas las ciudades se encuentren comunicadas entre si.

Figura 6.15 Pantalla de captura de Coordenadas.

Si los datos no son llenados con números positivos se obtiene el siguiente mensaje de error:

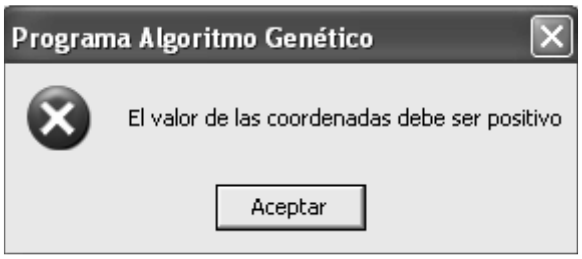


Figura 6.16 Pantalla de confirmación de captura de Coordenadas correcta

Del mismo modo se evalúa que se capturen sólo números:

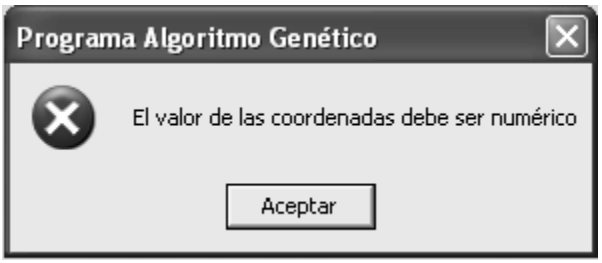


Figura 6.17 Validación de captura de sólo números.

Una vez que se han llenado correctamente los datos al oprimir en aceptar y se obtiene un mensaje que confirma el correcto llenado de la pantalla de captura. Al pulsar en **Aceptar** se vuelve a la pantalla principal donde ya se encuentran activos los botones de **Algoritmo Mejor Vecino** y **Algoritmo Genético**.

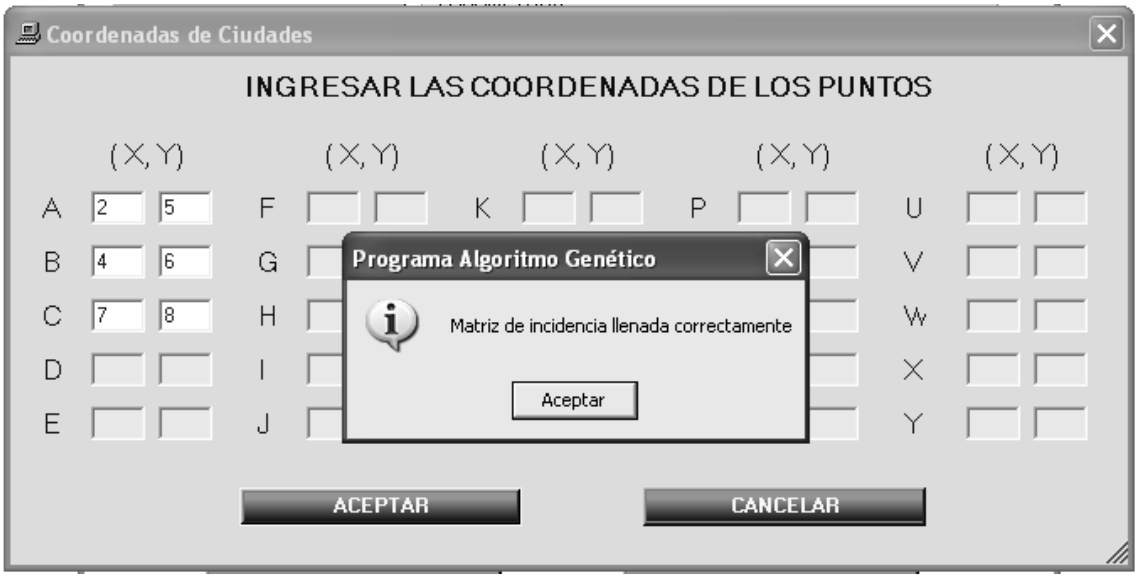


Figura 6.18 Pantalla de confirmación de captura de Coordenadas correcta.

Otra opción de captura de los datos consiste en el llenado directo de la matriz de adyacencia, consistente en la siguiente pantalla:

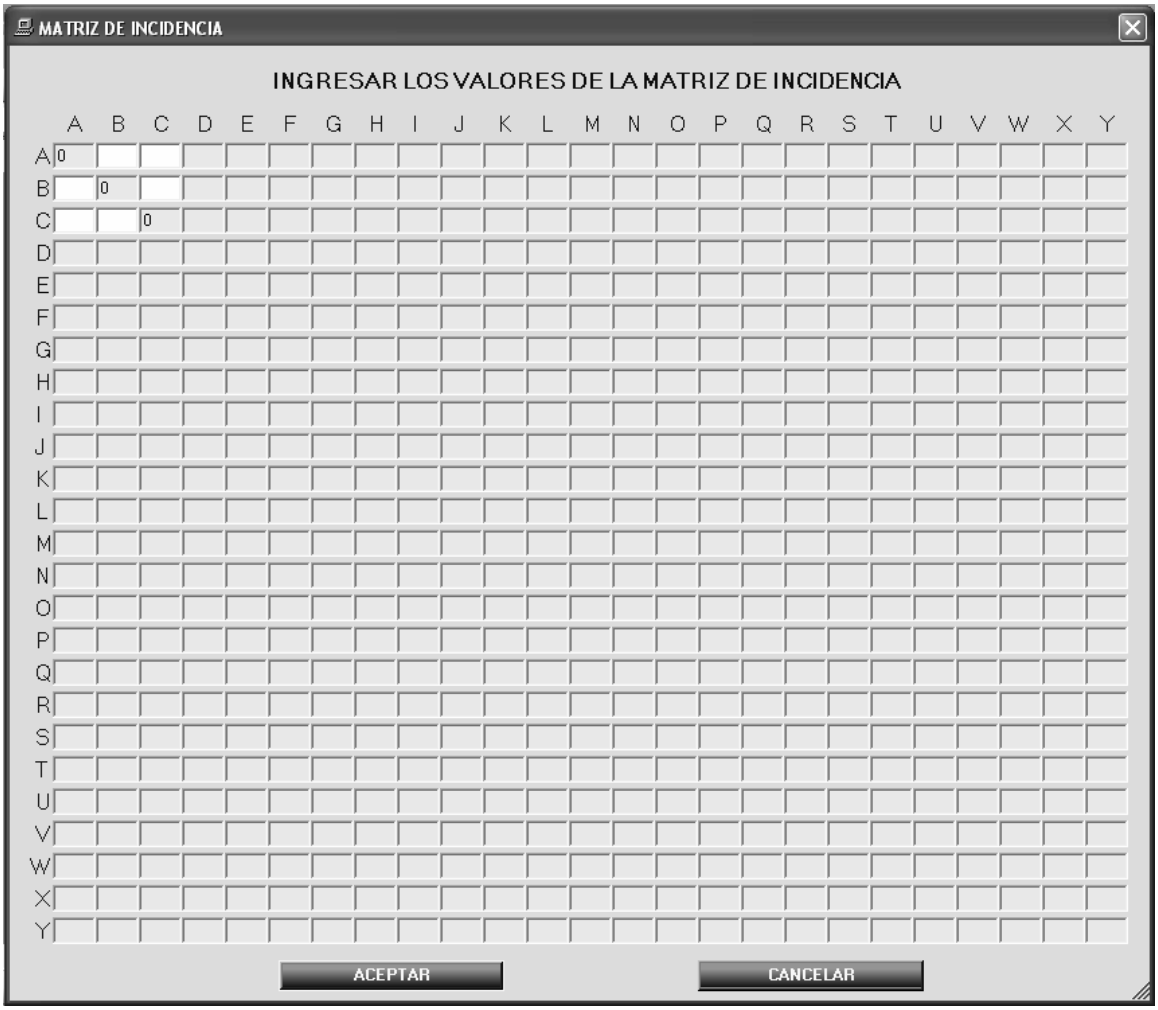


Figura 6.19 Pantalla de captura de Matriz de incidencia.

Se puede apreciar la matriz para su llenado manual, la cantidad de casillas activas está determinada por el número de ciudades que contiene el problema y que fueron indicadas en la pantalla principal, en él caso de la figura 6.19 fueron tres.

En el caso de no llenar la matriz correctamente, ingresando valores no numéricos, dejando espacios en blanco o debido a que no se respete la simetría se obtiene el correspondiente mensaje de error.

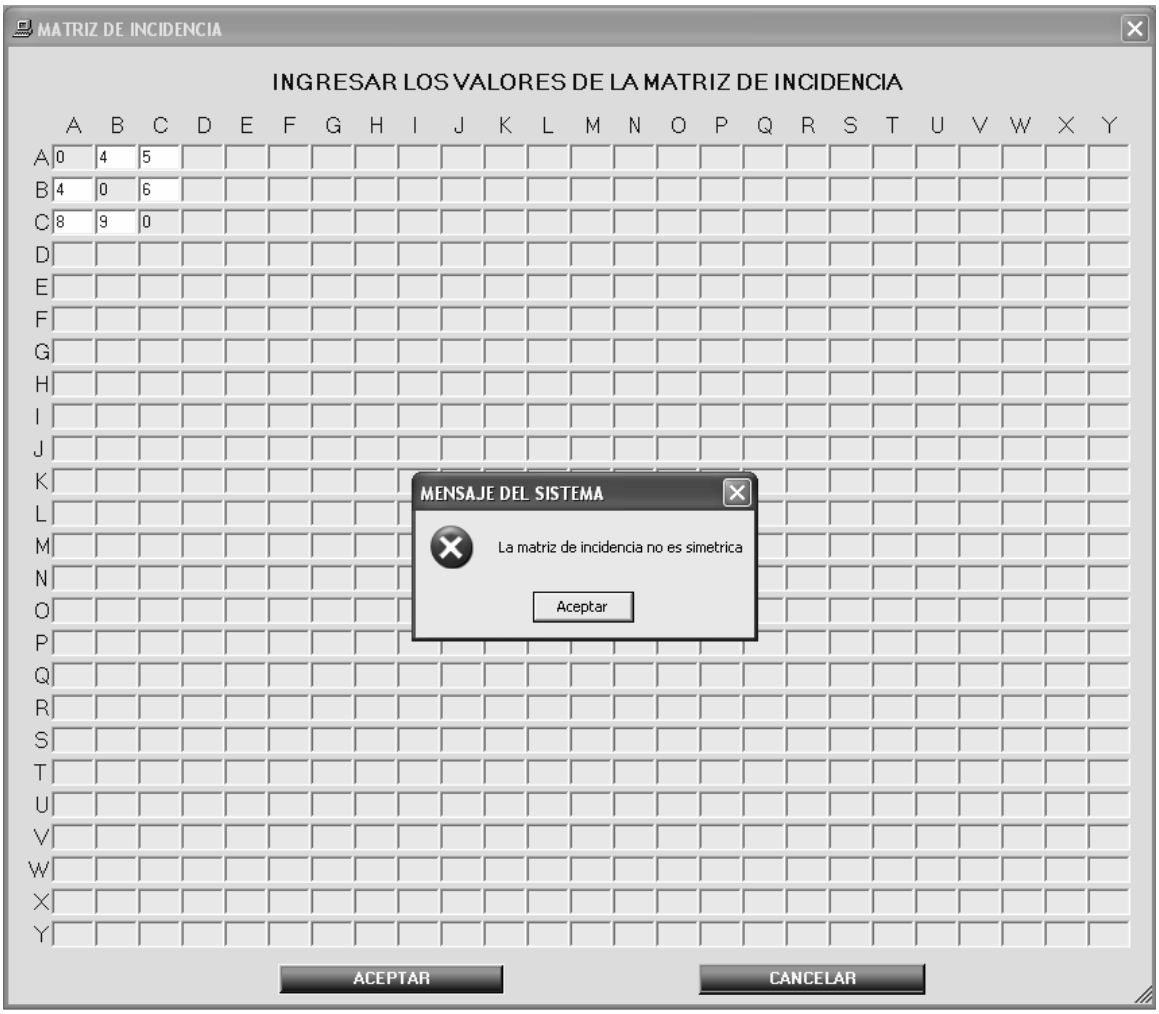


Figura 6.20 Validación de simetría de la matriz.

Una vez que se completa correctamente se obtiene la confirmación al dar clic en **Aceptar**.

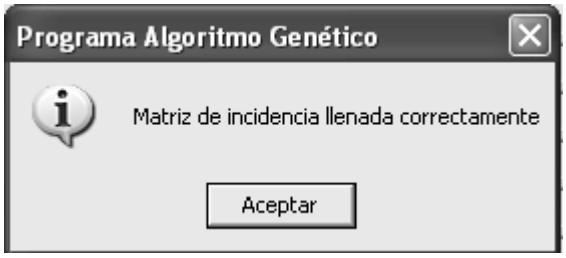


Figura 6.21 Confirmación de llenado correcto de la matriz.

Una vez que los datos se han llenado correctamente se regresa a la pantalla principal, donde se podrá realizar la ejecución de los dos algoritmos y observar los resultados

en la misma pantalla. Esto se realizará en la siguiente sección, donde se ejecutan los algoritmos con el caso de prueba acordado.

6.2 Prueba

Ahora se ejecuta el ejemplo elegido, primeramente se marcan los datos iniciales en la pantalla principal:

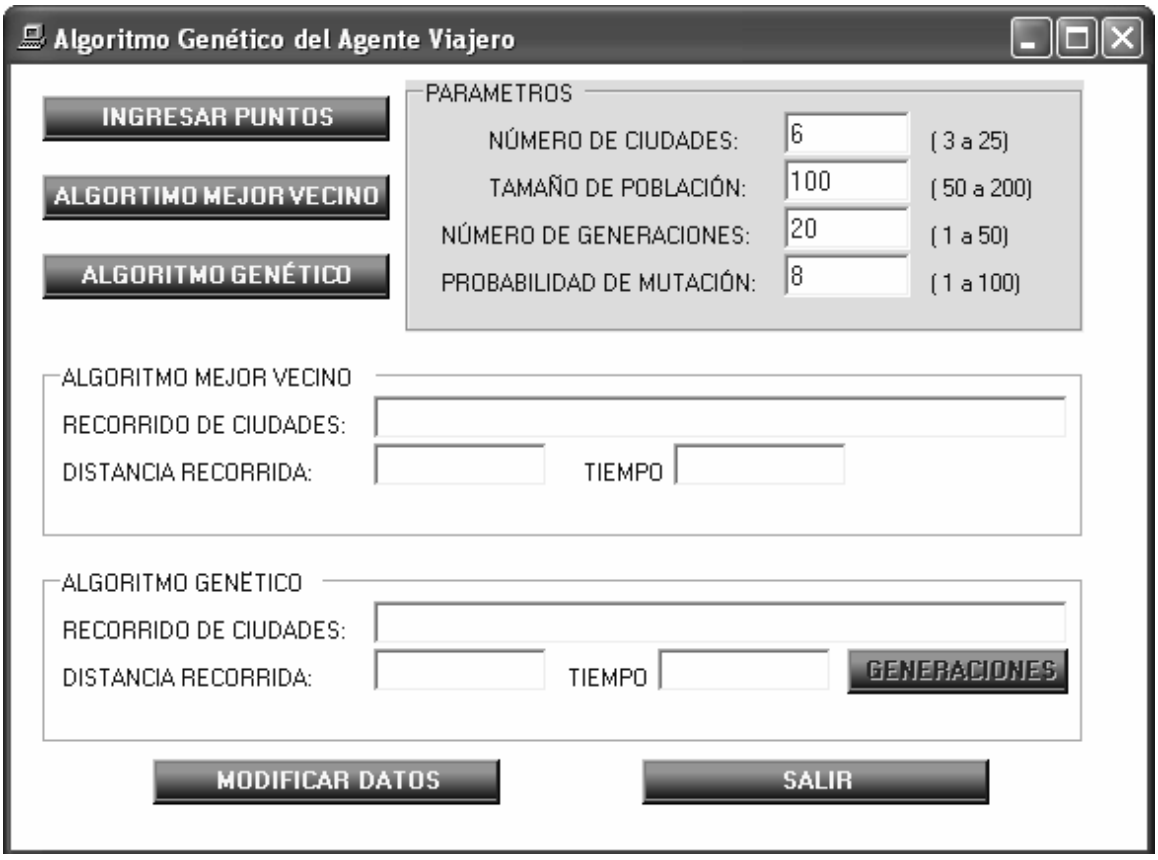


Figura 6.22 Llenado de los datos iniciales del problema.

Se eligió para esta prueba un número de 20 generaciones y un 8% de probabilidad de mutación, posteriormente se presentan los resultados obtenidos con otros parámetros.

Una vez completados estos datos es posible proceder a ingresar los valores de la matriz de incidencia de acuerdo con la matriz de incidencia del ejemplo acordado previamente al inicio del capítulo.

x,y	A	B	C	D	E	F
A	α	3	93	13	33	9
B	3	α	77	42	21	16
C	93	77	α	36	16	28
D	13	42	36	α	56	7
E	33	21	16	56	α	25
F	9	16	28	7	25	α

Figura 6.23 Matriz de incidencia con los datos del problema.

Figura 6.24 Pantalla con los datos capturados.

Una vez capturada la información de manera correcta se podrá proceder a la ejecución en la pantalla principal del programa del algoritmo del mejor vecino. Es de notar que la matriz respeta las reglas de llenado y es simétrica.

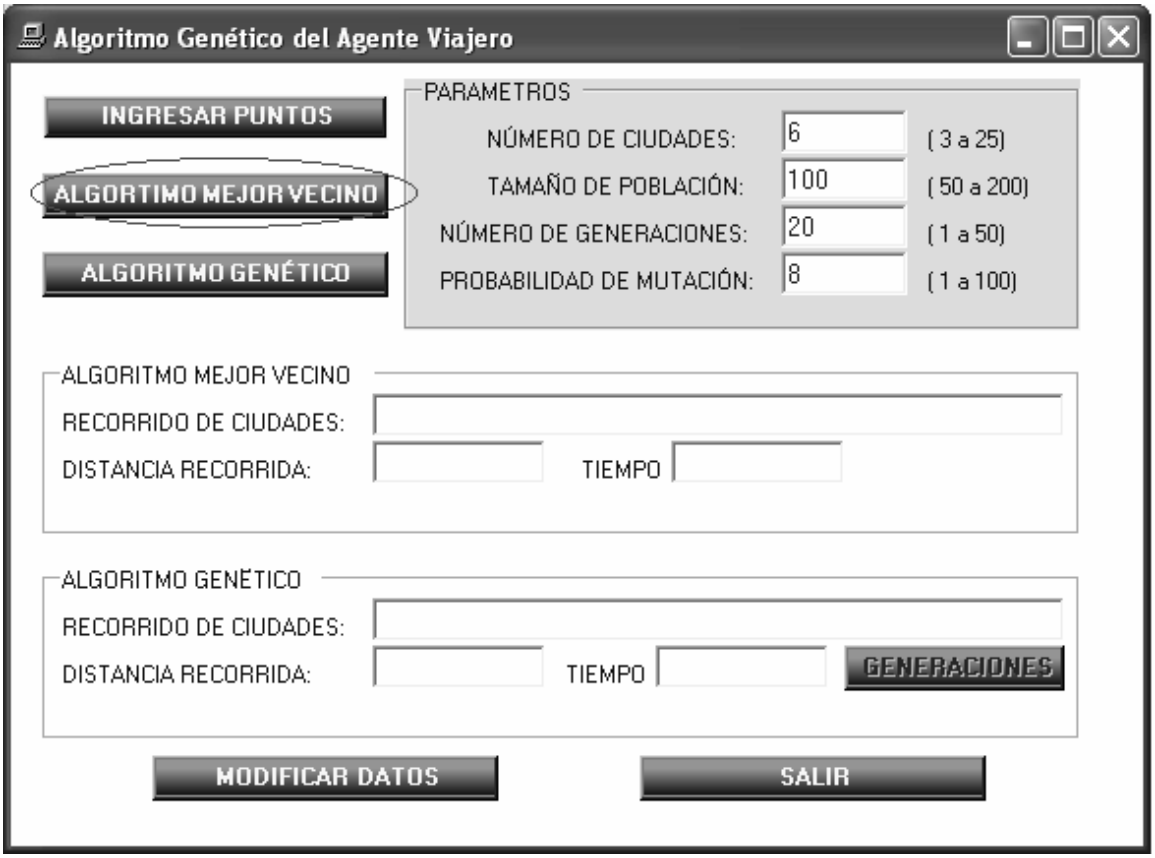


Figura 6.25 Pantalla principal ejecución del mejor vecino.

Al pulsar el botón **Algoritmo Mejor Vecino** se obtienen los resultados expuestos en la figura 6.26.

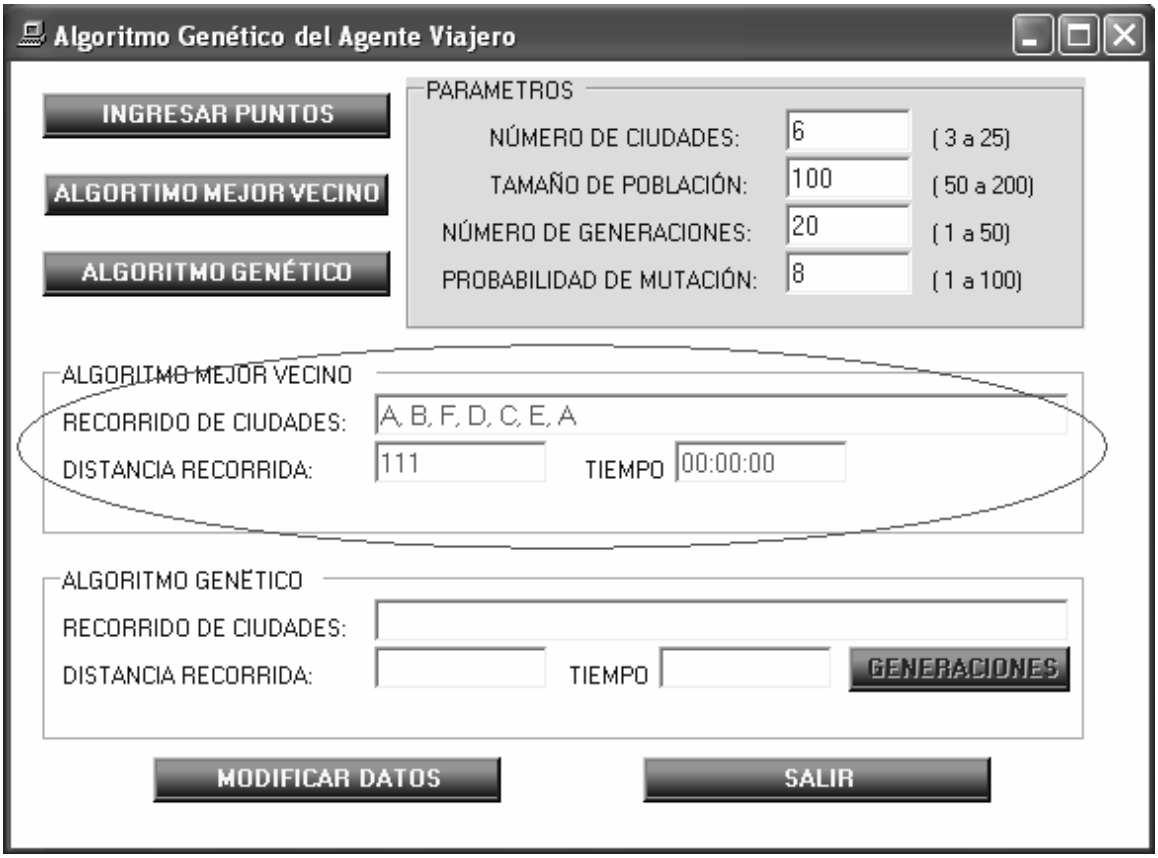


Figura 6.26 Pantalla principal con resultados del algoritmo Mejor Vecino.

Se observa que él tiempo de ejecución es muy corto y no alcanza a ser reflejado por el contador, ya que el tiempo invertido en encontrar esta solución es muy inferior a un segundo, en este caso inferior a las diezmilésimas de segundo. La parte de recorrido de ciudades índica la ruta de solución encontrada, mientras que las cifras en distancia recorrida informan el costo de realizar el recorrido por la ruta sugerida.

Se prueba ahora con la opción del algoritmo genético pulsando el botón como lo indica la figura 6.27 obteniendo los resultados mostrados en la figura 6.28, es de notar que debido a la naturaleza de los algoritmos genéticos los resultados serán variados al repetir la prueba, mientras que para él algoritmo mejor vecino se obtienen los mismos resultados en cada ejecución.

Figura 6.27 Pantalla principal de ejecución del algoritmo genético.

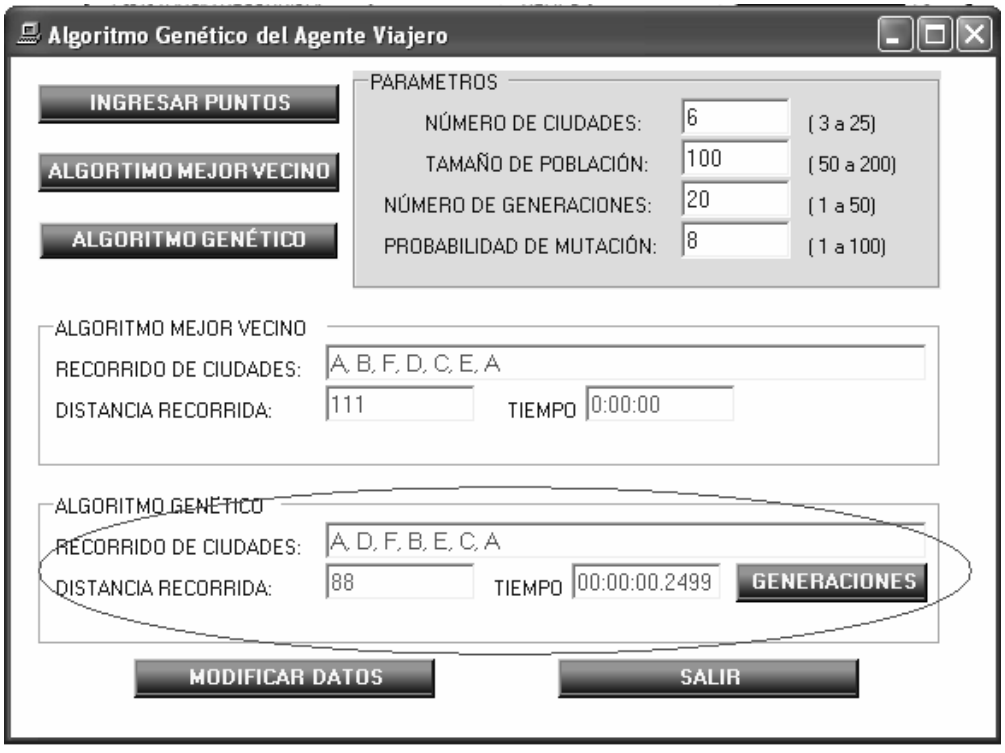


Figura 6.28 Pantalla principal con resultados del algoritmo genético.

Obsérvese que la ruta obtenida difiere de manera importante de la del algoritmo mejor vecino, si bien el tiempo de ejecución es más alto, la distancia recorrida desciende sensiblemente. Ahora bien el programa presenta sólo una solución en la pantalla principal y como se ha expuesto anteriormente los algoritmos genéticos trabajan con poblaciones y no con individuos, por lo que se selecciono la respuesta con el menor recorrido para mostrarla en pantalla, pero pueden existir más respuestas con idéntica puntuación pero diferente ruta, con esto en mente se le ha agregado a la pantalla principal el botón de **Generaciones**.

Figura 6.29 Botón de generaciones.

La funcionalidad de este botón consiste en mostrar en pantalla el contenido de un archivo de texto el cual guarda la información de las generaciones creadas por el algoritmo. Se muestra a continuación la pantalla de resultados:

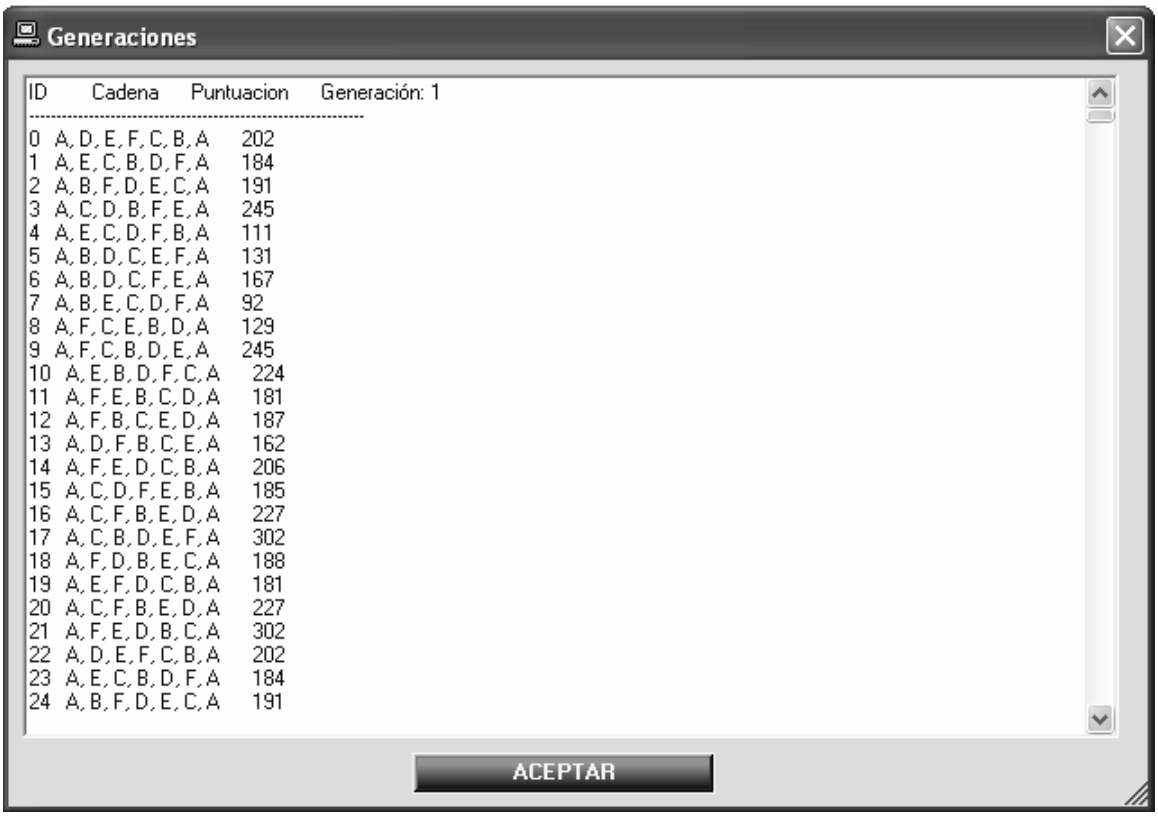


Figura 6.30 Generaciones.

Desplazándose por esta pantalla es posible observar la evolución del algoritmo genético. Primeramente se tiene el encabezado donde están indicados el identificador

(id) de cada individuo, la ruta de solución, la puntuación que índica el costo del recorrido y por último el número de generación. Al analizar los datos de la siguiente generación se percibe como el identificador de los individuos de la siguiente generación continua con números consecutivos a partir del último de la generación anterior; excepto por los primeros 5 individuos, los cuales ya existían en la generación anterior y pasaron a la siguiente por medio del elitismo.

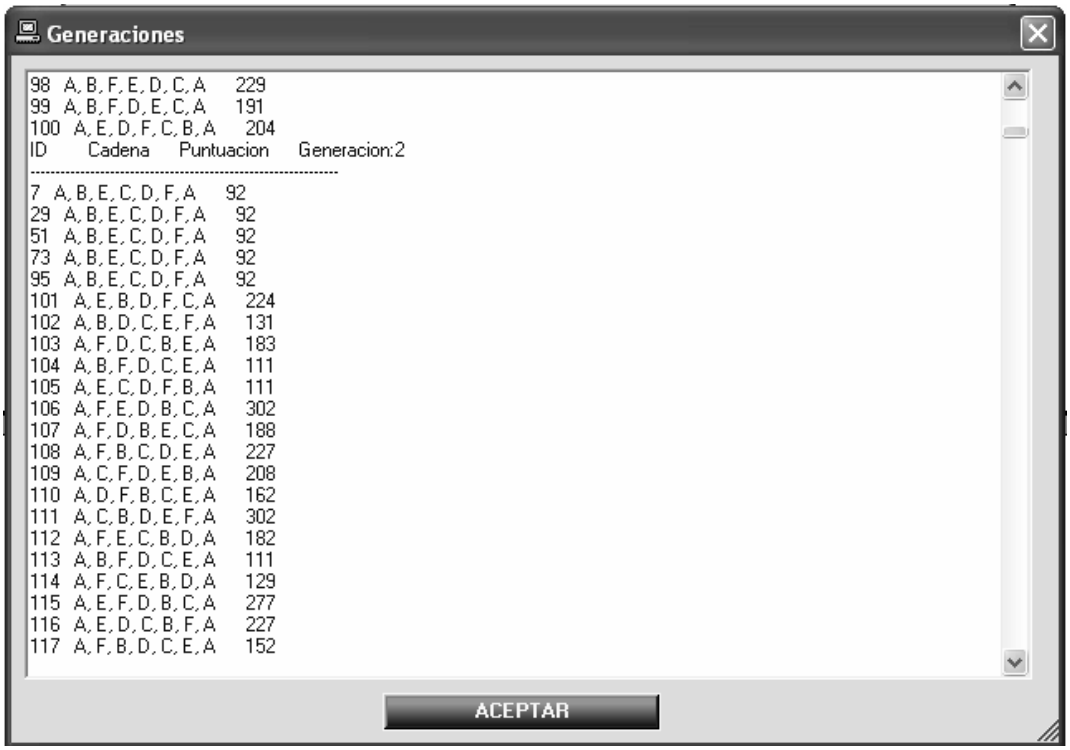


Figura 6.31 Generaciones siguientes.

De esté modo se puede observar la evolución de la solución hasta la generación final que en esté caso se alcanza al llegar al número de generaciones especificado, para esté caso fue de 20. La figura 6.32 muestra la imagen final en la siguiente página.

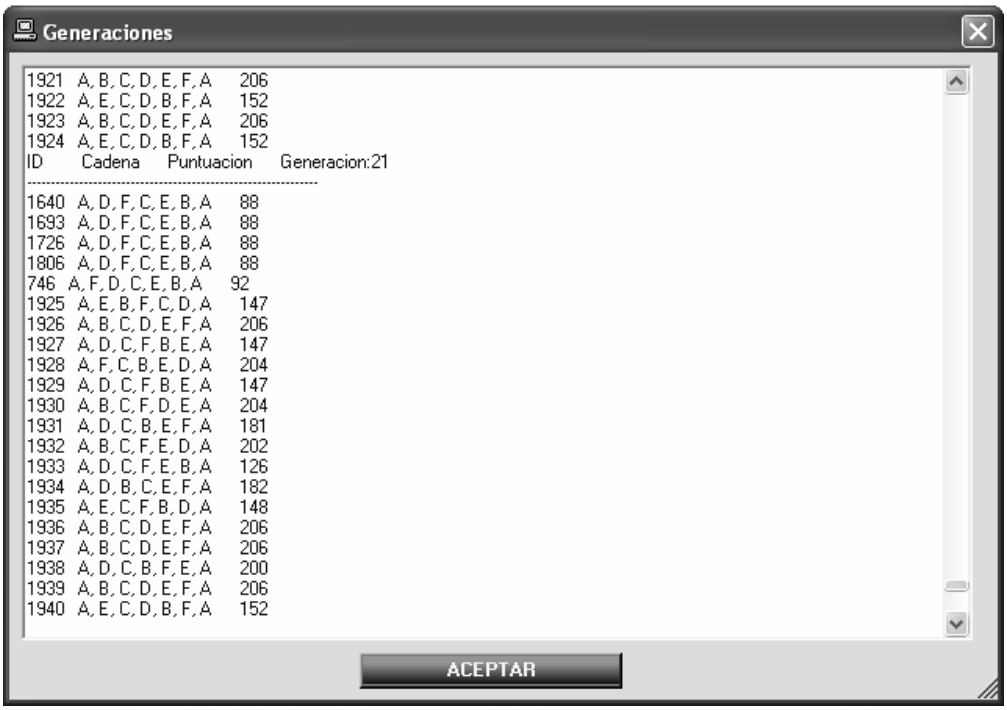


Figura 6.32 Generación Final.

Nótese que en este caso sólo el recorrido de solución A – D – F – C – E – B – A da el costo mínimo de 88. Es de suma importancia hacer notar que los algoritmos genéticos operan por medio del azar, si bien no son, como se ha explicado una búsqueda aleatoria; el hecho de que la generación de las primeras soluciones al azar, luego por cruce aunado al factor de la mutación sean los ejes del funcionamiento del algoritmo, induce un comportamiento diferente en cada ejecución, por lo que la repetición de este ejercicio podría generar o no diferentes resultados, es decir se puede llegar a la solución en menos generaciones, o no llegar en las 21 ocurridas en este caso. Se invita al lector a experimentar con los diferentes parámetros para observar los diversos comportamientos del algoritmo genético.

Aclarado esto se realiza él análisis de los resultados obtenidos en diversas ejecuciones del algoritmo genético y con diversos grupos de parámetros.

6.3 Interpretación de resultados.

Obsérvese los resultados de ambos algoritmos con auxilio de la tabla comparativa de la figura 6.33 en la siguiente página.

Parámetros a comparar	Algoritmo Mejor Vecino	Algoritmo Genético
Ruta seleccionada:	A – B – F – D – C – E – A	A – D – F – B – E – C – A
Distancia Recorrida:	111 Unidades	88 Unidades
Tiempo	00:00:00.0000	00:00:00.2499

Figura 6.33 Tabla comparativa de resultados de ambos algoritmos.

Debido a su naturaleza, a pesar de la repetición continua de la ejecución, el algoritmo del mejor vecino no presenta ninguna variación en sus resultados por lo que en las siguientes pruebas se muestra sólo una vez el resultado del mejor vecino y se enfoca la representación de datos al desempeño del algoritmo genético.

Para este ejemplo se tiene la siguiente matriz del problema del agente viajero con la cual se analizará su desempeño, comparativamente con el del mejor vecino mediante la medición de los tiempos de respuesta y mejor solución obtenida.

X,Y	A	B	C	D	E	F	G	H	I	J
A	∝	51	55	90	41	63	77	69	10	23
B	51	∝	10	64	8	53	10	46	73	72
C	55	10	∝	21	25	51	47	16	10	69
D	90	64	21	∝	2	9	17	5	26	42
E	41	8	25	2	∝	10	41	31	59	48
F	63	53	51	9	10	∝	17	47	32	43
G	77	10	47	17	41	17	∝	10	25	10
H	69	46	16	5	31	47	10	∝	10	24
I	10	73	10	26	59	32	25	10	∝	38
J	23	72	69	42	48	43	10	24	38	∝

Figura 6.34 Problema ejemplo.

Para el algoritmo del mejor vecino se obtuvieron los siguientes resultados:

Algoritmo mejor vecino:

Recorrido solución: A, I, C, B, E, D, H, G, J, F, A

Distancia total: 171

Tiempo solución: 00:00:00.0100

Para el algoritmo genético la solución obtenida con los siguientes parámetros es:

Tamaño de población	100
Número de generaciones	10
Probabilidad de mutación	5
Recorrido de ciudades	A, I, C, D, E, F, B, G, H, J, A
Distancia recorrida	105
Tiempo	00:00:00.4306

Figura 6.35 Solución.

Donde la solución óptima apareció en la quinta generación.

En diez sucesivas ejecuciones del algoritmo genético para los mismos parámetros se obtuvieron los siguientes resultados:

Ejecución 1	
Recorrido de ciudades	A, H, I, B, D, F, C, E, G, J, A
Distancia Recorrida	115
Tiempo	00:00:00.1802
Ejecución 2	
Recorrido de ciudades	A, I, H, C, D, F, B, E, G, J, A
Distancia Recorrida	127
Tiempo	00:00:00.1902
Ejecución 3	
Recorrido de ciudades	A, C, B, J, I, E, H, D, G, F, A
Distancia Recorrida	131
Tiempo	00:00:00.1201
Ejecución 4	
Recorrido de ciudades	A, J, G, B, E, C, D, F, H, I, A
Distancia Recorrida	151
Tiempo	00:00:00.1301
Ejecución 5	
Recorrido de ciudades	A, E, C, I, H, D, B, G, F, J, A
Distancia Recorrida	121
Tiempo	00:00:00.1301
Ejecución 6	
Recorrido de ciudades	A, E, B, C, D, F, I, G, J, H, A
Distancia Recorrida	155
Tiempo	00:00:00.1802
Ejecución 7	
Recorrido de ciudades	A, I, C, B, D, F, E, H, G, J, A
Distancia Recorrida	105

Tiempo	00:00:00.1201
Ejecución 8	
Recorrido de ciudades	A, I, E, B, H, C, D, G, F, J, A
Distancia Recorrida	191
Tiempo	00:00:00.1402
Ejecución 9	
Recorrido de ciudades	A, B, C, D, E, F, G, H, I, J, A
Distancia Recorrida	192
Tiempo	00:00:00.1201
Ejecución 10	
Recorrido de ciudades	A, I, H, C, B, E, D, F, G, J, A
Distancia Recorrida	115
Tiempo	00:00:00.1301

Figura 6.36 Tabla de Soluciones.

Es claro que los resultados son variables y se alcanzó el óptimo nuevamente en la séptima ejecución.

Alterando los parámetros y ejecutando nuevamente el algoritmo se obtienen los siguientes resultados.

Tamaño de población	150
Número de generaciones	20
Probabilidad de mutación	10
Recorrido de ciudades	A, C, I, J, H, D, E, G, F, B, A
Distancia recorrida	105
Tiempo	00:00:00.5708

Figura 6.37 Nuevos parámetros.

Ahora se presentan los resultados para las diez sucesivas ejecuciones siguientes:

Ejecución 1	
Recorrido de ciudades	A, F, D, B, I, E, C, G, J, H, A
Distancia Recorrida	123
Tiempo	00:00:00.3605
Ejecución 2	
Recorrido de ciudades	A, I, J, C, E, D, F, G, H, B, A
Distancia Recorrida	127
Tiempo	00:00:00.4806
Ejecución 3	
Recorrido de ciudades	A, G, C, B, J, H, F, I, D, E, A
Distancia Recorrida	123

Tiempo	00:00:00.3705
Ejecución 4	
Recorrido de ciudades	A, J, B, D, E, F, G, C, H, I, A
Distancia Recorrida	123
Tiempo	00:00:00.3605
Ejecución 5	
Recorrido de ciudades	A, D, G, C, J, B, E, F, I, H, A
Distancia Recorrida	121
Tiempo	00:00:00.3605
Ejecución 6	
Recorrido de ciudades	A, B, F, C, D, E, G, J, H, I, A
Distancia Recorrida	115
Tiempo	00:00:00.7711
Ejecución 7	
Recorrido de ciudades	A, C, D, B, E, F, G, H, I, J, A
Distancia Recorrida	122
Tiempo	00:00:00.7811
Ejecución 8	
Recorrido de ciudades	A, E, D, C, I, F, H, G, J, B, A
Distancia Recorrida	111
Tiempo	00:00:00.7811
Ejecución 9	
Recorrido de ciudades	A, I, E, J, B, D, F, C, H, G, A
Distancia Recorrida	115
Tiempo	00:00:00.9713
Ejecución 10	
Recorrido de ciudades	A, D, H, C, G, E, J, B, F, I, A
Distancia Recorrida	115
Tiempo	00:00:00.3605

Figura 6.38 Tabla de Soluciones.

El siguiente diagrama muestra los resultados en forma gráfica para el mejor vecino:

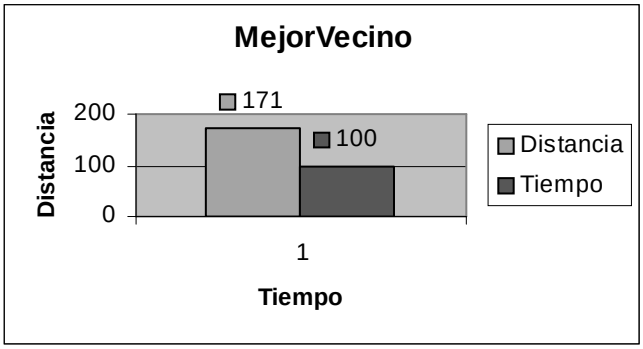


Figura 6.39 Gráfico Mejor Vecino.

La gráfica muestra que obtuvo una solución con un coste de 171 unidades de distancia en 100 milésimas de segundo.

Ahora se muestran los resultados del algoritmo genético con los primeros parámetros, para las 10 ejecuciones siguientes además de la inicial.

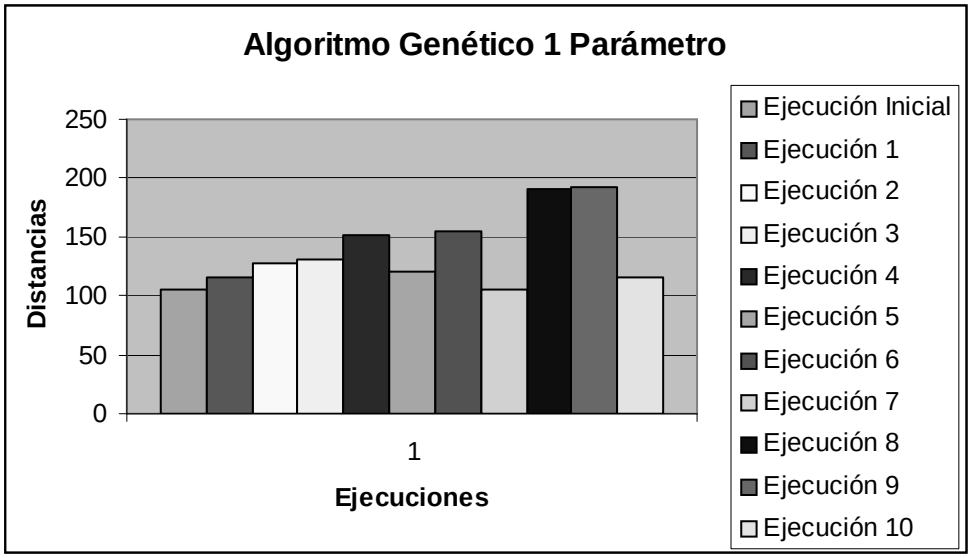


Figura 6.40 Gráfico Algoritmo Genético Distancias.

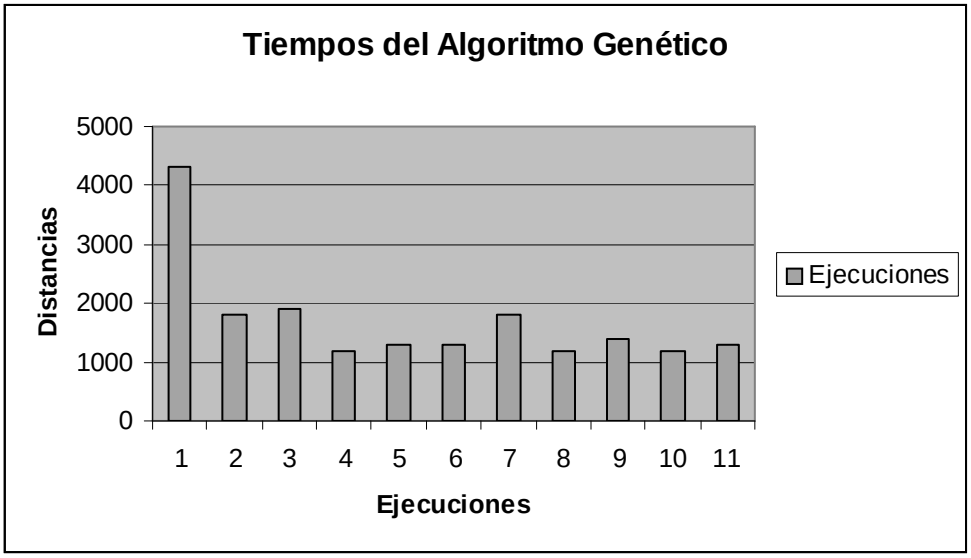


Figura 6.41 Gráfico Algoritmo Genético Tiempos.

Ahora se muestran los datos de los segundos parámetros para otras diez ejecuciones además de la inicial.

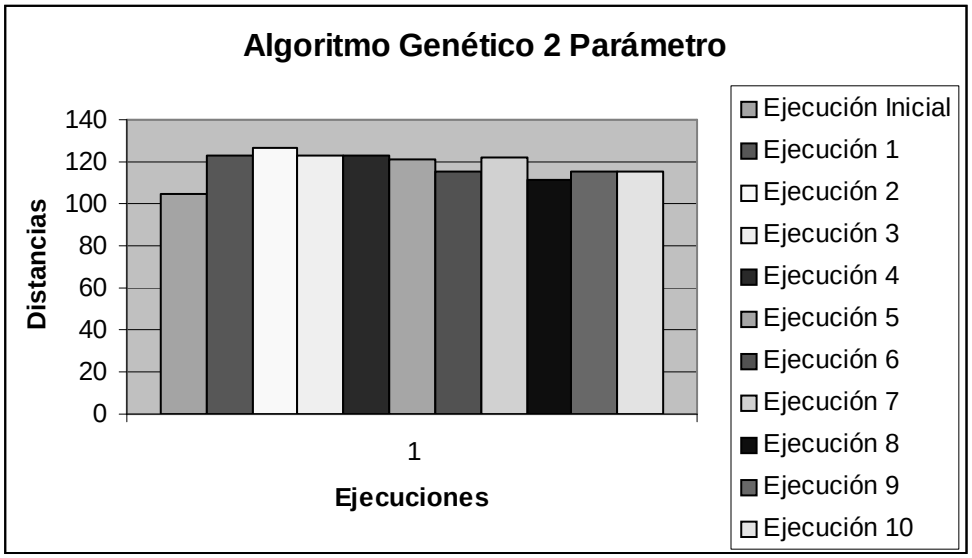


Figura 6.42 Gráfico Algoritmo Genético Distancias.

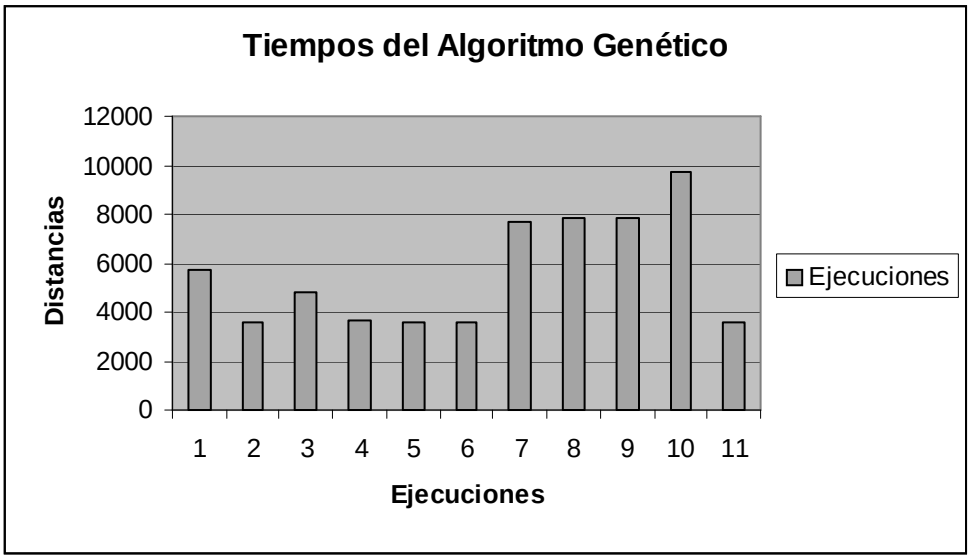


Figura 6.43 Gráfico Algoritmo Genético Tiempos.

Se aprecia que los resultados de la segunda ejecución son substancialmente más homogéneos y mejores respecto a los de la primera ejecución, si bien sus tiempos de ejecución son también más altos. Esto índica que la variación de los parámetros de entrada afectó considerablemente el desempeño del algoritmo. De manera que a mayor cantidad de individuos en la población, además de un mayor número de

generaciones y una mayor probabilidad de mutación mejoró el desempeño del algoritmo. En cualquier caso se obtuvieron mejores soluciones con el algoritmo genético que con el algoritmo del mejor vecino, es interesante también observar que la mutación es un factor de inestimable valor para conseguir la diversidad y evitar el estancamiento del algoritmo genético en unos pocos valores locales en apenas unas cuantas ejecuciones.

CONCLUSIONES

La propuesta presentada en este trabajo refrenda la idea de que el uso de los algoritmos genéticos es una alternativa práctica y funcional para resolver un problema de optimización como el del agente viajero para su caso simétrico.

La creación del algoritmo genético lleva a las siguientes conclusiones referentes al desarrollo del algoritmo:

El alfabeto utilizado formado por letras, permitió la correcta codificación del problema y con esto la creación de los operadores del algoritmo genético. La codificación de los genes fue descriptiva y permitió una buena representación aunque no se empleo la codificación basada en un alfabeto binario recomendada por el teorema de los esquemas.

La implementación de los operadores genéticos de cruce y mutación, se ha mostrado como excelentes generadores de diversidad y evitan el estancamiento del algoritmo, considerando que no es conveniente abusar de la mutación, ya que puede convertir al algoritmo en una búsqueda aleatoria. Sí es necesario su uso porque el operador de cruce, por si mismo lleva a unificar la población a un solo tipo de individuo e impide la evolución. La implementación de ambos operadores se logró con solamente ciertos ajustes al concepto original a fin de evitar la creación de individuos no viables.

La función de adaptación, fijada en la ruta de menor costo permite un criterio cuantificable de medición que sin lugar a duda, permite identificar que individuo es mejor que otro.

El criterio de selección de la ruleta consigue que el individuo con mejor función de adaptación, sea quien cuente con las mayores probabilidades de ser seleccionado como padre para el nuevo individuo de la siguiente generación. Mientras que la técnica del elitismo preserva a los mejores individuos, de manera que su material genético se conserve íntegro. Además de participar en la creación de la nueva generación por medio de la herencia, esto permite ahorrar tiempo al algoritmo; ya que al combinar el material genético de los individuos padre, se puede generar un nuevo individuo que se encuentre más lejos de la solución de lo que estaba al menos uno de sus padres, lo que no es deseable.

En cuanto al diseño del algoritmo genético la técnica presenta un bajo costo, tanto computacional como de implementación. Ejemplifica el proceso de traslado del concepto teórico de algoritmo, a la codificación en un lenguaje de programación orientado a objetos (POO). En este sentido la POO demuestra ser un paradigma ideal que permite, la creación de una clase llamada "*individuo*" con todos los métodos requeridos para la interacción de éste con la población y con el mismo propósito general del algoritmo.

El algoritmo genético demostró un comportamiento robusto; en cuanto a la comparación con el algoritmo del "*mejor vecino*" ya que tuvo un mejor desempeño, obteniendo buenos resultados en las pruebas y permitiendo realizar la evaluación de los parámetros que

afectan su desarrollo, tales como el tamaño de la población y la probabilidad de mutación.

Se puede decir que esta técnica presenta una buena alternativa de solución para problemas como el del Agente Viajero, y que el principal factor para el buen funcionamiento del algoritmo es mantener la diversidad durante el proceso evolutivo. Esta técnica evita la convergencia prematura y logra encontrar de manera consistente la zona factible de solución, con sólo mantener ciertas consideraciones en la implementación de los operadores sexuales que generan a los nuevos individuos. Además el algoritmo genético muestrea el espacio de búsqueda lo suficiente para obtener resultados competitivos a un costo computacional y de implantación bajo, frente a métodos exhaustivos. El no requerir un conocimiento profundo del problema para poder encontrar su solución, proporciona una clara ventaja a los algoritmos genéticos frente a otros enfoques.

Los conocimientos adquiridos durante la carrera son los cimientos que han permitido el desarrollo de este trabajo de tesis. La comprensión teórica y matemática del algoritmo, es parte fundamental de la tesis. Los conocimientos prácticos de diseño en lenguajes de programación de alto nivel con representación gráfica basada en UML, diagramas de flujo de datos y pseudocódigo; son técnicas adecuadas para la implantación de algoritmos en software de aplicación. En este sentido el trabajo de tesis, liga ambos aspectos de la carrera, cumpliendo con el objetivo de complementar la formación académica del profesional, en la Licenciatura de Matemáticas Aplicadas y Computación.

TRABAJO FUTURO

Posibles complementos a este trabajo son los siguientes:

- Refinar el algoritmo resolviendo los empates por medio de criterios que permitan un mejor desempeño de éste.
- Buscar nuevas alternativas para mantener la diversidad sin la necesidad de agregar más parámetros al algoritmo.
- Adaptar el algoritmo a problemas del agente viajero que no sean simétricos.
- Analizar la factibilidad de acoplar esta técnica con algún algoritmo de optimización de otro tipo (tal vez un método heurístico).

BIBLIOGRAFÍA

1. **ACKOFF**, Russell. Fundamentos de la investigación de operaciones. Tr Enrique Jiménez. México, Ed. Limusa 1982 502 p.
2. **ASIMOV** Isaac. Introducción a la ciencia. Biblioteca de divulgación científica Muy interesante. 1985. 427 p.
3. **BALENA** Francesco Programación Avanzada con Visual Basic .NET Tr Jorge Rodríguez Vega México Ed Mc Graw Hill, 2002 1238 p.
4. **BRONSON**, Richard, Investigación de operaciones. Tr María de Lourdes Fournier, Ed Mc Graw Hill, 1989, 324p.
5. **DAELLENBACH**, Hans. Introducción a las técnicas de investigación de operaciones. 3 ed, Tr Lourdes Fournier. México, Ed Continental, 1990, 711 p.
6. **DARWIN** Charles, El origen de las especies. Tr Aníbal Froufe Ed RBA 2002, (c 1983) 493 p.
7. **EDWARD** M Reingold, Jurg Nievergelt, Narsing Deo. Combinatorial Algorithms Theory and Practice; Prentice-Hall, Inc. Englewood Cliffs. New Jersey 1977.
8. **GARDUÑO** Rivera Alan. Simulación de procesos evolutivos mediante algoritmos genéticos. Edo de México, 1998. 113 p. Tesis (Licenciatura en Matemáticas aplicadas y Computación) UNAM ENEP-A
9. **GAREY** Michael / Jonson David. Computers and Intractability A Guide to the theory of NP- Completeness W H Freeman and Co. New York 1979.
10. **GILLETT**, Billy. Introduction to operations researchs. USA. Ed Mc Graw Hill, 1976 (c 1976) 606 p.
11. **GOLDBERG** David Genetic algorithms in searchs, Optimization and machine learning. USA Ed Addison-Wesley, 1989.
12. **HALVORSON** Michael Visual Basic .NET. Tr Jorge Rodriguez Vega México Ed Mc Graw Hill, 2002 651 p.
13. **HERRAN** Manuel de la. Algoritmos genéticos avanzados. Revista Solo Programadores #37. Ed. Towercom. Septiembre 1997.
14. **HOLLAND** John H, Adaptation in natural and artificial systems. University of Michigan Press. 1975.
15. **PEÑAFIEL**, Luis. Programación lineal. Ed Trillas México 1982, 229 p.
16. **PRAWDA**, Juan. Métodos y modelos de investigación de operaciones Vol. 1 Ed Limusa 1993, 935 p.
17. **REYES** Ángel. Tesis: Un ejemplo de Optimización de Funciones por Medio del Algoritmo Ariad's Clew: El caso del Problema del Agente Viajero. UDLA 1997.
18. **ROSS** Kenneth y Wright Charles. Matemáticas Discretas. Prentice Hall 1990.
19. **SIERRA** Molina Guillermo. Sistemas expertos en contabilidad y administración de empresas. Ed RA-MA. 1995.
20. **STINSON** D R. An Introduction to the Design and Analysis of Algorithms; 2 ed. Winnipeg Manitoba Canada 1987.

21. **SWOKOWSKI**, Earl. Cálculo con Geometría Analítica. Grupo Editorial Iberoamericana 1989, México Tr José Luis Abreu y Martha Oliveró. 1097 p.
22. **VARELA** Hernández Maria Del Rosario. Tesis Diseño de una Heurística para el tratamiento del Problema del Agente Viajero. UDLA 1996.
23. **VITE** Pérez Herlinda. Computación evolutiva. Edo de México, 1998. 100 p. Tesis (Licenciatura en Matemáticas aplicadas y Computación) UNAM ENEP-A
24. **WILF** H S Algorithms and Complexity Prentice Hall 1986.
25. **ZBIGNIEW** Michalewicz. Genetic Algoritms + Data Structures = Evolution Programs. Springer-Verlag, 3 Ed, 1996.

ENLACES DE INTERNET

1. Algoritmos genéticos.
   http://www./si.upc.es/~iea/transpas/9_geneticos/sld001.html
2. Algoritmos genéticos en Visual Basic.
   http://geneura.urg.es/~jmerelo/DegaX/menu.html
3. Algoritmos genéticos.
   http://www.itnuevolaredo.edu.mx/takeyas/Algoritmos_Geneticos/
4. Computación evolutiva. Anselmo Pérez.
   ftp://ftp.de.uu.net/pub/research/softcomp/EC/EA/papers/intro-spanish.ps.gz
5. GAIA Artificial Life.
   http://www.geocities.com/SiliconValley/Vista/7491/
6. Genetic Algorithms FAQ,
   http://www.cs.cmu.edu/Groups/Al/html/faqs/ai/genetic/top.html
7. Introducción a los algoritmos genéticos,
   http://hp.fciencias.unam.mx/revista/soluciones/N17/Coello2.html
8. Introducción a los algoritmos genéticos de Carlos Coello.
   http://www.redcientifica.com/doc/doc/999042600/l.html
9. Manú Herran Page.
   http://www.aircenter.net/gaia/
10. Red científica.
    http://www.redcientifica.com/cgi-bin/index/index.pl
11. Teoría de la evolución.
    http://fai.unne.edu.ar/biologia/evolucion/evo1.htm
12. The traveling salesman problem,
    http://csvax.cs.caltech.edu/~pepe/cs20/a/lab3/
13. Traveling Salesman Problem Home Page,
    http://www.mbhs.edu/class/compmethods/iwilliam/tsp.html
14. Tutorial informática evolutiva.
    http://geneura.urg.es/~jmerelo/ie/index.html
15. Universidad del Valle de México.
    http://sacbeo.8m.com/ligastutoriales/index.html
16. Introducción a los algoritmos genéticos. Francisco José Ribadas Peña
    http://ccia.ei.uvigo.es/docencia/IA/practicas/seminarioAG.pdf
17. Complejidad Algorítmica Juan Luís Castro Peña. Depto. Ciencias de la Computación e Inteligencia Artificial. Universidad de Granada. 1999.
    http://decsai.ugr.es/~castro/CA/node24.html