INSTITUTO TECNOLÓGICO DE LA PAZ
DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN
MAESTRÍA EN SISTEMAS COMPUTACIONALES

UN ALGORITMO MEMÉTICO PARA LA OPTIMIZACIÓN EN PARALELO DEL TSP

T E S I S

QUE PARA OBTENER EL GRADO DE
MAESTRO EN SISTEMAS COMPUTACIONALES

PRESENTA:
JOEL ARTEMIO MORALES VISCAYA

DIRECTOR DE TESIS:
DR. MARCO ANTONIO CASTRO LIERA

LA PAZ, BAJA CALIFORNIA SUR, MÉXICO, JUNIO 2016

SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO
Instituto Tecnológico de La Paz

La Paz, B.C.S., 15/junio/2016

DEPI/257/2016
Asunto: Autorización de impresión.

C. JOEL ARTEMIO MORALES VIZCAYA,
ESTUDIANTE DE LA MAESTRÍA EN
SISTEMAS COMPUTACIONALES,
P R E S E N T E.

Con base en el dictamen de aprobación emitido por el Comité Tutorial de la Tesis denominada: **"UN ALGORITMO MEMÉTICO PARA LA OPTIMIZACIÓN EN PARALELO DEL TSP",** mediante la opción de tesis (Proyectos de Investigación), entregado por usted para su análisis, le informamos que se **AUTORIZA** la impresión.

A T E N T A M E N T E
"Ciencia es Verdad, Técnica es Libertad"

M.A.T.I. LUÍS ARMANDO CÁRDENAS FLORIDO,
JEFE DE LA DIV. DE EST. DE POSGRADO E INV.

INSTITUTO TECNOLÓGICO DE LA PAZ
DIVISIÓN DE ESTUDIOS DE POSGRADO
E INVESTIGACIÓN

C.c.p. Coordinación de la Maestría.
C.c.p. Depto. de Servicios Escolares.

LACF/fkso*

SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

Tecnológico Nacional de México
Instituto Tecnológico de La Paz

DICTAMEN DEL COMITÉ TUTORIAL

SUBDIRECCIÓN ACADÉMICA
DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN.

La Paz, B.C.S., 15 /junio/ 2016

C. MATI. LUIS ARMANDO CÁRDENAS FLORIDO,
JEFE DE LA DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN,
P R E S E N T E.

Por medio del presente, enviamos a usted dictamen del Comité Tutorial de tesis para la obtención del grado de Maestro, con los siguientes datos generales:

No. de Control M14310025	Nombre JOEL ARTEMIO MORALES VISCAYA
Maestría en:	SISTEMAS COMPUTACIONALES
Título de la tesis: UN ALGORITMO MEMÉTICO PARA LA OPTIMIZACIÓN EN PARALELO DEL TSP	
DICTAMEN: Se autoriza el trabajo de investigación, en virtud de que realizó las correcciones correspondientes conforme a las observaciones planteadas por este Comité Tutorial.	

A t e n t a m e n t e .
El Comité Tutorial

MC. JESÚS ANTONIO CASTRO

MSC. ILIANA CASTRO LIERA

DR. MARCO ANTONIO CASTRO LIERA

c.c.p. Coordinador de la Maestría.
c.c.p. Departamento de Servicios Escolares.
c.c.p. Estudiante.

Resumen En este trabajo convergen el problema del agente viajero, el aprovechamiento del cómputo paralelo y la aplicación de algoritmos heurísticos.

El problema del agente viajero, más conocido como TSP (*Traveling Salesman Problem),* es uno de los más estudiados en el área de optimización combinatoria. La resolución de instancias de este problema es de fundamental importancia en áreas de la ingeniería, tan diversas como: logística, fabricación de circuitos integrados, tableros de circuitos perforados, medición de rayos X, data clustering, mira de telescopios, conexión de antenas, entre otros.

Los algoritmos heurísticos se han venido utilizando para resolver problemas de optimización y búsqueda en los que no existen métodos eficientes para resolver todas sus instancias, cómo es el caso de TSP. Los algoritmos heurísticos, aunque no pueden garantizar encontrar la mejor solución, nos proporcionan soluciones razonablemente buenas en tiempos cortos.

El uso eficiente de los recursos computacionales es un aspecto importante en el desarrollo de algoritmos de optimización, particularmente en lo que respecta al aprovechamiento de arquitecturas de cómputo paralelo, ya que, prácticamente todos los modelos de computadoras modernas cuentan con más de una unidad de procesamiento.

El punto central de esta tesis es la propuesta de un nuevo algoritmo memético aplicado al problema del agente viajero, con una estrategia de búsqueda local que mezcla características de 3-OPT y Lin-Kernighan. El algoritmo se implementó en un cluster de computadoras. Se incluyen resultados estadísticos de su desempeño al aplicarlo en diferentes instancias obtenidas de la TSPLIB, con tamaños de entre 100 y 500 ciudades. Además se demuestra el rendimiento superior del algoritmo propuesto al de otras heurísticas como ACO, AG, 2-OPT y 3-OPT.

Abstract The present work deals with the traveling salesman problem, the use of parallel computing, and the application of heuristic algorithms.

The traveling salesman problem or TSP, is one of the most intensively studied problems in combinatorial optimization, it has applications in several areas such as: logistics, electronic boards manufacturing, X-ray measuring, data clustering, astronomy, antenna connections, among others.

Heuristic algorithms are used for producing good solutions to optimization and search problems in a reasonable time frame, but the found solutions may not be optimal. Usually, heuristic algorithms are used for hard problems that cannot be solved by classical means or when there are not efficient algorithms for solving all their instances, that's the case of TSP.

When we are dealing with hard optimization problems like TSP, where the search space grows exponentially, we need to use the computational resources available as efficiently as possible, particularly it is of great importance to harness the power of parallel computing, because most modern computers have multiple processing units.

The central matter of this work, is the design and implementation of a new memetic algorithm applied to TSP. The algorithm uses a local search inspired on 3-OPT and the Lin-Kerninghan strategy. The algorithm was implemented in a computer cluster and the results were compared with those produced by other widely-used algorithms for TSP solving (ACO, AG, 2-OPT and 3-OPT). The instances of the problem used for experimentation where taken from TSPLIB with sizes of 100 to 500 cities.

Índice general

Índice de figuras III

Índice de tablas IV

1. Introducción 1
- 1.1. Motivación . . . 1
- 1.2. Definición del problema . . . 3
- 1.3. Objetivo general . . . 4
- 1.4. Objetivos específicos . . . 4
- 1.5. Justificación . . . 4

2. El problema del agente viajero 6
- 2.1. Optimización . . . 6
  - 2.1.1. Optimización combinatoria . . . 7
- 2.2. El problema del agente viajero . . . 8
  - 2.2.1. Formulación formal . . . 9
  - 2.2.2. TSPLIB . . . 10
- 2.3. Complejidad computacional . . . 10
  - 2.3.1. Problemas clase P y NP . . . 11
  - 2.3.2. Complejidad de TSP . . . 13

3. Cómputo paralelo 14
- 3.1. Memoria distribuida . . . 15
  - 3.1.1. MPI . . . 16

4. Algoritmos existentes para TSP 17
- 4.1. Introducción . . . 17
- 4.2. Algoritmos de aproximación . . . 19
  - 4.2.1. Algoritmo de Christofides . . . 20
- 4.3. Programación lineal y cotas . . . 21
- 4.4. Algoritmos heurísticos para el TSP . . . 24
  - 4.4.1. Algoritmos de mejoras iterativas . . . 26
  - 4.4.2. Algoritmos poblacionales . . . 28
  - 4.4.3. Algoritmos genéticos . . . 29
    - 4.4.3.1. Selección . . . 29
    - 4.4.3.2. Cruza . . . 30
    - 4.4.3.3. Mutación . . . 31

4.4.3.4. Características de un AG 31
4.4.4. Algoritmos genéticos paralelos 32
4.4.5. Algoritmos meméticos 34

5. Estrategia propuesta **37**
5.1. Algoritmo de búsqueda local propuesto 40

6. Pruebas y Resultados **43**

7. Conclusiones y trabajo futuro **47**
7.1. Conclusiones 47
7.2. Trabajo futuro 48

8. Bibliografía **49**

Anexo: Código fuente **54**
8.1. Código principal (viajero.c) 54
8.2. Cabeceras 63
8.2.1. Cabecera con la implementación de la búsqueda local 63
8.2.2. Cabecera con los parámetros del programa, params.h 71
8.2.3. Cabecera para la generación de aleatorios, alea.h 71

Índice de figuras

2.1. Problema de optimización con varios mínimos locales y un mínimo global. 7
2.2. Grafo completo con ocho vértices 9

4.1. Ejemplo del algoritmo de Christofides 21
4.2. Radios de control 23
4.3. Peor caso, radios de control 23
4.4. *Moats* para rellenar espacios 23
4.5. a. Heurística basada en trayectorias, b. poblacional 25
4.6. Intercambio de las aristas (b,e) y (f,c) por las aristas (b,c) y (f,e) 27
4.7. Cruza para cadenas binarias 30

5.1. Diagrama de flujo de un AG con migración 38
5.2. Diagrama de flujo del algoritmo memético propuesto 39
5.3. Ejemplo de un cambio *double bridge* 40
5.4. Ejemplo de iteración posible con reemplazo de tres aristas 41

Índice de tablas

6.1. Comparación de algoritmos de búsqueda local 44
6.2. Comparación de Heurísticas 44
6.3. Parámetros del Sistema de Hormigas Paralelo 45
6.4. Parámetros del AG Distribuido 45
6.5. Parámetros del algoritmo memético 45
6.6. Resultados experimentales instancias Churritz y Reinelt (TSP225). 45
6.7. Resultados experimentales instancias Krolak/Felts/Nelson. 46
6.8. Resultados experimentales instancias Padberg/Rinaldi. 46

Capítulo 1

Introducción

1.1. Motivación

La optimización se puede definir como la ciencia encargada de encontrar las mejores soluciones a los problemas, que generalmente modelan una realidad física.

En el día a día tratamos con decenas de problemas de optimización, los cuales resolvemos muchas veces sin percatarnos de que lo estamos haciendo; son problemas simples que pueden ser resueltos de manera analítica o con simple observación, sin embargo, entre más grande es el abanico de posibilidades y mayor la cantidad de variables a considerar, los problemas se van volviendo más complejos y es entonces que necesitamos algoritmos y herramientas que nos ayuden a solucionarlos.

El uso eficiente de los recursos computacionales es un aspecto cada vez más importante en el desarrollo de algoritmos de optimización, particularmente en lo que respecta al aprovechamiento de arquitecturas de cómputo paralelo, ya que en la actualidad prácticamente todos los modelos de computadoras personales cuentan con procesadores multi-núcleo o con más de un procesador, por lo que el desarrollo de aplicaciones en paralelo ya no está restringido a grandes y costosos equipos especializados.

Existen además una gran cantidad de programas o librerías que facilitan la conexión de varios equipos en una red, que se comportan como una sola computadora en paralelo, lo que se conoce como *cluster*, así como tecnologías que permiten el desarrollo de aplicaciones de propósito general en paralelo utilizando los procesadores de las tarjetas gráficas.

La optimización numérica ha sido ampliamente usada para resolver problemas en áreas como la economía, ingeniería de control, arquitectura, investigación de operaciones y mecatrónica, por

nombrar solo algunas.

Existen dos tipos de problemas de optimización: de variables continuas, dónde el conjunto de búsqueda tiene cardinalidad infinita (entre cualesquiera dos soluciones aceptables se puede encontrar una tercera solución aceptable), y los problemas de variables discretas, también llamados de combinatoria, en los cuales el número de soluciones posibles es finito o numerable; éste es el tipo de problemas en el que nos vamos a enfocar.

Muchos problemas del mundo real son difíciles de resolver, algunas de las razones por las que pueden serlo son [1]:

- El número de posibles soluciones en el espacio de búsqueda es tan grande, que es imposible realizar una búsqueda exhaustiva.
- El problema es tan complicado que, sólo para encontrar una respuesta, tenemos que utilizar modelos tan simplificados del problema que cualquier resultado obtenido es inservible.
- La función de evaluación que describe la calidad de cualquier solución propuesta tiene ruido o varía en el tiempo, por lo que no buscamos una solución sino una serie completa de soluciones.
- Las soluciones factibles están sujetas a tantas restricciones que construir una sola solución es difícil.
- La persona que resuelve los problemas no está adecuadamente preparada o tiene alguna barrera psicológica que le impide encontrar una solución.

Lógicamente, esta lista puede extenderse con muchos otros obstáculos posibles, pero, en el contexto del presente trabajo estos son suficientes.

Los principales obstáculos de los algoritmos convencionales es que son demasiado demandantes computacionalmente, es decir requieren recursos significativamente grandes en tiempo y almacenamiento. En algunos problemas combinatorios altamente complejos, los algoritmos heurísticos basados en poblaciones han sido aplicados con éxito.

Se conoce como heurísticos a todos aquellos métodos que solamente necesitan la definición de un espacio de soluciones, los operadores y la función objetivo, sin exigir el cumplimiento de

ninguna característica especial, como podría ser la continuidad, la existencia de derivadas, etc. La importancia vital de estos métodos está en que son la única posibilidad para resolver problemas con las condiciones siguientes:

- No existe un algoritmo eficiente para encontrar una solución particular.
- No es posible la exploración exhaustiva del espacio completo de búsqueda, debido a las dimensiones del mismo.

Ninguno de los métodos heurísticos garantiza la obtención del óptimo global en el marco de condiciones prácticas. Sin embargo, permiten obtener buenas soluciones que en muchos casos satisfacen a los usuarios [2].

El método que se propone en este trabajo es una combinación de un algoritmo de búsqueda global basado en poblaciones y un algoritmo de búsqueda local realizada por cada individuo. Mientras el algoritmo global heurístico explora el espacio completo de búsqueda, el algoritmo local explota determinadas regiones del mismo espacio, a fin de refinar las soluciones obtenidas, lo que se conoce como un algoritmo "Memético" [3].

1.2. Definición del problema

Si un vendedor inicia un viaje (recorrido) en la ciudad donde vive y tiene que visitar cada una de las ciudades de una lista de su itinerario, exactamente una vez y regresar a casa, él podría elegir una ruta o recorrido al azar. No obstante, si el vendedor quiere ahorrar tiempo y energía, por no mencionar el gasto de combustible, tratará de encontrar la ruta más corta con la cual pueda cumplir el objetivo de su trabajo. Si se conoce la distancia que existe entre cada par de ciudades, es posible encontrar el camino más corto sumando las distancias de cada ruta posible y después comparándolas. Sin embargo, cuando el número de ciudades aumenta, el tiempo que se requiere para calcular la ruta más corta también se incrementa. La cuestión de encontrar el recorrido más corto ha sido bautizada con el nombre de "El problema del agente viajero" o TSP por sus siglas en inglés (*Traveling Salesman Problem*).

1.3. Objetivo general

Desarrollar un algoritmo de optimización para el TSP simétrico, basado en estrategias heurísticas y un algoritmo de búsqueda local, que sea capaz de mejorar la eficacia y eficiencia de los algoritmos heurísticos existentes, y que pueda ser ejecutado en arquitecturas paralelas de bajo costo.

1.4. Objetivos específicos

- Diseñar un algoritmo memético para optimizar TSP.
- Implementar el algoritmo memético en una arquitectura paralela de bajo costo (*cluster* de computadoras).
- Contar con un análisis estadístico comparativo de la eficacia y eficiencia del algoritmo desarrollado, resolviendo problemas definidos en TSPLIB contra otras estrategias heurísticas representativas.

1.5. Justificación

TSP es un problema importante, pues constituye un componente central de un número considerable de problemas del mundo real, además de aquellos para los que fue descrito originalmente (problemas de rutas, transportes y logística), que se pueden formular como TSP.

Algunas de las aplicaciones usuales son las siguientes:

- Fabricación de circuitos integrados. El problema es minimizar el tiempo que se requiere al grabar una secuencia de lineas (aristas) que forman las conexiones eléctricas entre los diferentes componentes de un circuito integrado. Para ello, se utiliza una máquina que graba las líneas según el diseño. Ésta se mueve a la posición donde principia la interconexión y traza una línea, terminando en una posición diferente, para después continuar con la siguiente y así sucesivamente. De esta manera, los puntos de inicio y final del grabado corresponden a las ciudades en el TSP [4].

- Tableros de circuitos perforados. Para permitir el acoplamiento de los circuitos integrados y otros componentes, es necesario hacer orificios de diferentes diámetros. La broca que realiza los orificios se debe cambiar cuando todos los orificios de un diámetro específico han sido terminados. El problema se puede ver como una serie de TSP donde las ciudades son los puntos a ser taladrados y la distancia es el tiempo necesario para moverse de una posición a otra [4].

- Medición de rayos X. Un aparato especial mide la intensidad de la radiación que se refleja en diferentes posiciones. Para obtener una medición precisa, las lecturas deben de superar las 30'000 posiciones. Con base en lo anterior, los puntos de lectura representan las ciudades, y el tiempo que se requiere para moverse de un punto a otro es la distancia entre las ciudades [4].

Capítulo 2

El problema del agente viajero

El problema del agente viajero es un problema NP-duro dentro de la optimización combinatoria en el area de mátematicas. El presente capítulo pretende aclarar estos conceptos.

2.1. Optimización

La tendencia a optimizar es inherente al ser humano, pues le resulta altamente motivante la idea de poder hacer más con menos. Esto le ha llevado a buscar formas más creativas y controladas de realizar sus actividades.

En general, para problemas sencillos es suficiente la observación y la experiencia para llegar a soluciones óptimas; pero cuando el problema se hace más complejo porque intervienen una mayor cantidad de variables o se incrementa el espacio en el que se va a buscar la solución, es necesario realizar un análisis más detallado, lo cual aumenta la cantidad de operaciones y hace necesario refinar el método para resolver el problema o utilizar herramientas que faciliten la tarea.

Optimizar implica determinar los valores para una serie de parámetros de tal manera que bajo ciertas restricciones se satisfaga alguna condición como: el más grande, el más pequeño, el menos caro, etcétera.

Formalmente, un problema de optimización se define como el conjunto de todas las instancias de dicho problema, donde una instancia I se define por un par (A, f) donde A se conoce como el espacio de búsqueda y f es la función de costo en problemas de minimización, o de beneficio en

problemas de maximización, que se utiliza para determinar la calidad de cada solución en A.

La optimización se aplica en muchos campos y especialidades e implica tareas de minimización o maximización. Dado que encontrar el máximo de una función f es equivalente a encontrar el mínimo de -f , los métodos pueden usarse para ambos tipos de problemas. Igualmente, debido a que existe una estrecha relación entre los problemas de optimización y búsqueda, se suelen utilizar indistintamente uno u otro término.

2.1.1. Optimización combinatoria

Un problema de optimización puede ser representado de la siguiente forma: dada una función $f : A \rightarrow \Re$, buscar un elemento $x_0 \in A \mid f(x_0) \leq f(x) \ \forall \ x \in A$ (minimización), donde A es conocido como espacio de búsqueda.

Si A es un subconjunto continuo de $\Re^n$ donde n es la dimensión del problema, estamos hablando de un problema de optimización en espacios continuos u optimización continua. Por otro lado, si A es discreto, es decir los elementos $x \in A$ tienen una cantidad finita o numerable de configuraciones distintas, estamos ante otra clase de problemas, conocidos como de optimización combinatoria.

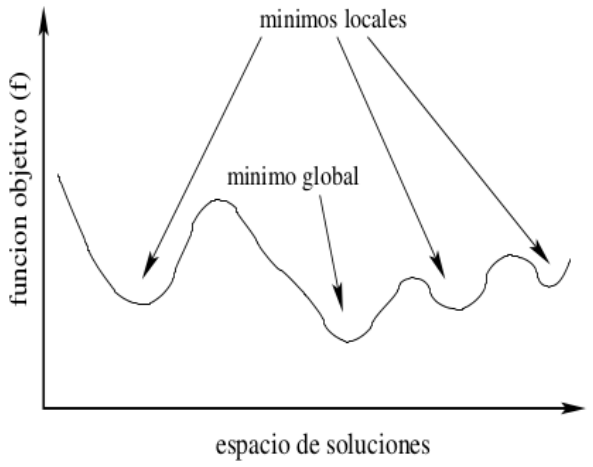

Figura 2.1: Problema de optimización con varios mínimos locales y un mínimo global.

La Fig. 2.1 muestra gráficamente un problema de optimización con varíos mínimos locales (elementos en el dominio de A que son mínimos en algún subconjunto de A) y un mínimo global correspondiente a la solución óptima del problema.

La optimización combinatoria tiene numerosas aplicaciones a problemas que se presentan en la industria, logística, ciencias, ingenierías y en la administración de organizaciones. Como ejemplos podemos mencionar, entre otros, el enrutamiento y carga de vehículos en redes de distribución, el diseño de redes de telecomunicación, la planificación de la producción, la selección de carteras financieras, la asignación de tareas a procesadores, el análisis de estructuras moleculares, las subastas de frecuencias para radiotransmisión, la asignación de tripulaciones en líneas aéreas, la planificación de la generación de electricidad y la distribución de ambulancias en una región para asegurar un cierto nivel de servicio a su población. La gran variedad de posibles aplicaciones es una fuente constante de problemas nuevos para la investigación en esta área [5].

2.2. El problema del agente viajero

Un viajante quiere visitar n ciudades una y sólo una vez cada una, empezando por cualquiera de ellas y regresando al mismo lugar del que partió. Supongamos que conoce la distancia entre cualquier par de ciudades. ¿De qué forma debe hacer el recorrido si pretende minimizar la distancia total?.

Debido a que todas las ciudades deben ser visitadas una vez, el problema se reduce a encontrar el orden en el cual se deben visitar dichas ciudades. Cada solución candidata se puede representar por medio de una permutación p de tamaño n que representa el orden en el que las ciudades deben ser visitadas. El costo C de las soluciones candidatas es la sumatoria de las distancias entre cada par de ciudades y se calcula por medio de (2.1).

$$C = (\sum_{i=1}^{n-1} D_{p_i,p_i+1}) + D_{p_n,p_1} \tag{2.1}$$

A este problema se lo conoce con el nombre de problema del agente viajero o TSP y resulta ser uno de los más prominentes dentro del campo de la optimización combinatoria, pues todavía no ha sido resuelto eficientemente. Para poder estudiarlo mejor es necesario entender algunos conceptos de teoría de grafos.

Entendemos por grafo al par $G = (V, E)$, donde V es un conjunto finito de elementos, que

llamamos vértices, y E es un conjunto de pares de vértices que denominamos aristas. Un ciclo es una sucesión de vértices $u_1, u_2, ..., u_p$ tales que $u_1, u_2, ..., u_p$ son distintos, $u_p = u_1$ y $u_i, u_{i+1} \in E$. Si el ciclo pasa por cada arista una y sólo una vez, se conoce como euleriano o de Euler, quien trabajó con este tipo de ciclos al resolver el problema de los puentes de Königsberg [6]. Si además, contiene a todos los vértices, se llama hamiltoniano (en honor al matemático irlandés Sir William Rowan Hamilton). Se conoce como cubrimiento a un conjunto de aristas sin vértices en común, y se dice que es perfecto si dicho recubrimiento contiene todos los nodos. Existen grafos que poseen propiedades destacables. Por ejemplo, decimos que un grafo es completo si cada par de vértices están unidos por una arista, es decir, que contiene todas las posibles aristas, como se ve en la Fig. 2.2

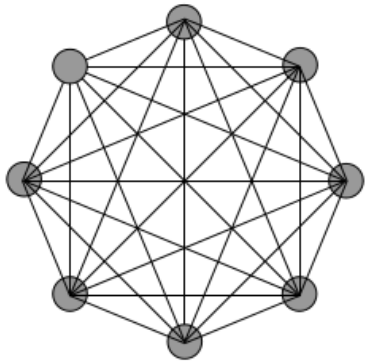

Figura 2.2: Grafo completo con ocho vértices

Para un grafo que tiene asignado para cada una de sus aristas un cierto peso, el TSP es equivalente a encontrar un ciclo hamiltoniano de mínimo peso, entendiéndose por peso del ciclo a la suma de los pesos de todas las aristas que pertenecen a él.

2.2.1. Formulación formal

Matemáticamente podemos pensar al TSP como el problema de hallar un ciclo hamiltoniano de mínima distancia en un grafo completo $G = (V, E)$ en donde $V = 1, 2, ..., n$ es el conjunto de nodos y representan las ciudades, E es el conjunto de aristas que denotan la conexión entre ellas y $d : E \to \Re_+$ una función que a cada $(i, j) \in E$ le asigna la distancia d_{ij} entre las ciudades i y j [7].

Algunos casos particulares de este problema son:

- Problema del agente viajero simétrico, si $d_{ij} = d_{ji} \ \forall \ (i,j) \in E$
- Problema del agente viajero asimétrico, si $\exists (i,j) \in E \mid d_{ij} \neq d_{ji}$
- Problema del agente viajero triangular, si $d_{ik} \leq d_{ij} + d_{jk} \ \forall \ i,j,k$

Existen también multiples extensiones al problema, entre las más comunes se encuentran:

- Problema del agente viajero múltiple m-TSP, en el cual hay m agentes viajeros, y se busca un conjunto de rutas que cubran todos los vértices [8].
- Problema del agente viajero multicolor MTSP, en el cual hay vértices de k colores distintos, y el número de vértices visitados entre cada dos visitas a vértices de mismo color está acotado [9].

2.2.2. TSPLIB

TSPLIB es una biblioteca con ejemplos de instancias de TSP (y problemas derivados) de varias fuentes y tipos, que incluye instancias tanto de TSP simétrico como del problema del ciclo hamiltoniano, el TSPD (responder sólo si el costo mínimo es menor a una constante), entre otros [10]. Es mantenida por el alemán Gerhard Reinelt, de la Universidad de Heidelberg, desde 1995.

La biblioteca contiene un archivo con las especificaciones para cada instancia, con su nombre, el tipo de problema, las dimensiones o tamaño del problema, comentarios adicionales y, debido a que no todas las instancias proporcionan una matriz de distancias entre ciudades como tal sino una ubicación, proporciona además una descripción de la métrica que debe utilizarse para calcular las distancias en dicha instancia.

Para todas las instancias de TSP simétrico, TSPLIB proporciona además el costo y recorrido óptimo.

2.3. Complejidad computacional

Empezaremos dando la noción de lo que es un problema. Un problema es una pregunta general a responder. Usualmente posee varios parámetros o variables libres, cuyos valores no son especifi-

cados. Podemos plantearlo dando una descripción general de todos sus parámetros y declarando las propiedades que la respuesta o solución debe satisfacer. Una instancia se obtiene especificando valores particulares para todos los parámetros del problema. Si, por ejemplo, tomamos al TSP simétrico, los parámetros serán el conjunto finito de ciudades y para cada par de ciudades, la distancia entre ellas.

Se dice que un algoritmo resuelve un problema π si produce siempre una solución para cualquier instancia I de π. Llamamos entrada a la descripción de una instancia que le damos al algoritmo, lo que podemos hacer a través de una palabra, que es una secuencia finita de símbolos elegidos de un alfabeto finito. Hay diferentes formas de describir una misma instancia de un problema. El tamaño de la entrada para una instancia I se define como el número de símbolos en la descripción de I. Al tamaño de la entrada se le usa para definir formalmente el tamaño de la instancia.

Desde el punto de vista computacional, es muy importante poder predecir el tiempo que le va a llevar a un algoritmo resolver una cierta instancia de un problema. Dicho tiempo por lo general puede ser expresado en términos de una sola variable: el tamaño de la instancia. Por ejemplo, lo que contribuye a la cantidad de información de la entrada en una instancia de m ciudades del TSP será el número de ciudades y los $m(m-1)/2$ números que definen las distancias. La complejidad de un algoritmo M para una entrada x de tamaño n es $C_{M(n)} = max_{x=n}$ { número de pasos que realiza M para procesar x }. Se dice, además, que un algoritmo M es polinomial si existe un polinomio P tal que $C_M(n) \leq P(n)\ \forall\ n$.

2.3.1. Problemas clase P y NP

En lo que se refire a complejidad computacional, existen cuatro grandes categorías de problemas, la clase P con el conjunto de todos los problemas que pueden ser resueltos por un algoritmo polinomial, es decir, que para cada problema $\pi \in P$, existe un algoritmo y un polinomio p de forma que una instancia de π cuya entrada es de tamaño w puede ser resuelto por ese algoritmo en a lo sumo $p(w)$ pasos.

Los problemas P son considerados tratables, sobre todo cuándo las instancias tienen un tamaño no muy grande, la multiplicación por ejemplo, es fácil y se clasifica como un problema tipo P.

Aunque los números por multiplicar vayan siendo mayores, el tiempo requerido para calcular el resultado aumenta de una manera relativamente modesta (polinomial).

Por otra parte, decimos que π pertenece a la clase NP si para cualquier instancia de entrada w existe una solución propuesta $v(w)$ y un algoritmo que con una entrada (v, w) demuestra en un tiempo polinomial que v resuelve π, es decir, la clase NP contiene todos aquellos problemas para los que existe un algoritmo que comprueba una solución en un tiempo polinomial, es fácil ver que $P \subset$ NP, pues suponiendo $\pi \in P$, por definición hay un algoritmo que dada cualquier instancia w $\in \pi$ lo demuestra en tiempo polinomial.

La factorización algebráica es un ejemplo de un problema NP, ya que a pesar de que no existe un algoritmo que lo resuelva en tiempo polinomial (no pertenece a la clase P), comprobar una solución candidata es trivial, suponga que desea factorizar 428,783; pese a que no se conoce un algoritmo eficiente para encontrar divisores, si alguien le diera un papel con los números 521 y 823, podría comprobar en segundos que son los divisores correctos.

Algunos expertos creen que si es posible verificar rápidamente soluciones a un problema de decisión, debe ser posible también obtener respuestas rápidamente (en tiempo polinómico). La pregunta aún no resuelta ¿es P=NP?, se considera el problema más importante en el campo de la complejidad computacional, al grado que el Clay Mathematics Institute ha ofrecido un premio de un millón de dólares estadounidenses para quién desarrolle la primera demostración correcta [11].

Para definir una tercera categoría de problemas, los llamados NP-Completos necesitamos primeramente introducir el concepto de reducibilidad polinomial. Supongamos que queremos resolver el problema de decisión A y que tenemos un algoritmo que resuelve el problema de decisión B. Supongamos también que tenemos un algoritmo que construye una instancia y de B para cada instancia x de A de tal forma que un algoritmo para B responde si para y solamente cuando la respuesta al problema A para x es si, y viceversa. Dicho algoritmo se denomina algoritmo de transformación. El algoritmo de transformación combinado con el algoritmo para B nos da un algoritmo para A.

Si existe un algoritmo de transformación polinomial del problema de decisión A en el problema de decisión B, se dice que el problema A es reducible polinomialmente al problema B, y se denota

como $A\alpha B$. Decimos entonces que B es un problema más duro que A. Un problema B es NP-Completo si está contenido en NP y para algún otro problema $A \in$ NP, $A\alpha B$.

Existe una última clase, denominada NP-Duro, que consta de todos aquellos problemas H que satisfacen que $\forall\ L \in$ NP, $L\alpha H$, es decir, cualquier problema NP puede reducirse al problema H, si existiera un algoritmo polinomial para resolver un problema NP-Duro, todos los demás podrían reducirse al mismo y tendrían un algoritmo polinomial.

2.3.2. Complejidad de TSP

Richard M. Karp demostró en 1972 [12] que el problema de determinar si en un grafo existe un ciclo hamiltoniano es NP-Completo. Esto resulta relevante, ya que dicho problema es un caso especial de TSP porque si le asignamos a todas las ramas del grafo peso 0 y a las ramas faltantes las agregamos asignándole peso 1, se tendrá otro grafo que tiene ciclos hamiltonianos. Resolviendo el TSP, el nuevo grafo tendrá un ciclo hamiltoniano de mínimo peso = 0 si y sólo si el grafo original contiene un ciclo hamiltoniano lo cual implica que TSP es un problema NP-Duro y por lo tanto, es probable que el tiempo de ejecución para cualquier algoritmo que lo resuelva aumente, en el peor caso, de forma exponencial con respecto al número de ciudades.

Capítulo 3

Cómputo paralelo

El cómputo paralelo consiste en la ejecución de múltiples instrucciones de un programa simultáneamente en varias unidades de procesamiento.

Originalmente una máquina paralela se definía como computadora con dos o más procesadores para ejecutar varias instrucciones de manera simultánea. El poder de computación de estas máquinas podía incrementarse aumentando su número de procesadores, en vez de reemplazar cada procesador con otro más complejo [13].

Desafortunadamente, aunque las computadoras paralelas con un par de procesadores han disminuído de precio, para la mayoría de las instituciones públicas todavía es complicado contar con estas máquinas; sin embargo, los equipos personales ya cuentan con procesadores multinúcleo y algunos más también con tarjetas gráficas con capacidad de cómputo de propósito general [14].

Los programas computacionales paralelos son más difíciles de escribir que los secuenciales, porque la concurrencia introduce nuevos tipos de errores de software, siendo la comunicación y sincronización entre diferentes subtareas algunos de los mayores obstáculos para obtener un buen rendimiento de un programa paralelo [15]. También resulta importante controlar las porciones de código en las que se accede a recursos compartidos o que no pueden ser paralelizadas, ya que se requiere su ejecución para poder llevar a cabo alguna de las subtareas. Ésta porción de código se conoce como "sección crítica" y se busca que sea lo más pequeña posible.

3.1. Memoria distribuida

Una alternativa para implementar la programación paralela son los sistemas de memoria distribuida, que consisten en un conjunto de nodos interconectados en red que colaborativamente intercambian y procesan información. Los nodos son unidades independientes con procesador y memoria local. Solo los procesadores locales tienen acceso directo a la memoria local en cada nodo. Por tanto, si un nodo requiere acceder a la memoria de otro nodo, se hace con un modelo de comunicación de paso de mensajes. Dentro del conjunto de nodos, a uno de ellos se le asigna la tarea de iniciar, distribuir tareas y recopilar resultados de ejecución de la aplicación paralela. Este nodo se conoce como maestro [13].

A esta alternativa de programación paralela se le denomina *cluster* de computadores y es muy utilizado en instituciones educativas, debido a que:

- Cada uno de los nodos es una máquina que se puede utilizar como una estación de trabajo normal, y en sus tiempos muertos como un componente del *cluster*.
- La construcción de una supercomputadora con una cantidad elevada de procesadores es compleja y cara; sin embargo, crear una red con una cantidad elevada de nodos es mucho más sencillo y barato.
- El *cluster* puede ser altamente tolerante a fallos ya que, con el diseño adecuado de aplicaciones, la desconexión de uno de los nodos no impide que el resto del *cluster* siga trabajando.
- Escalar una arquitectura de este tipo es sumamente sencillo, ya que sólo se deberán incluir más estaciones de trabajo al *cluster*.
- Existen diversas herramientas de desarrollo libres y gratuitas que permiten la implementación de un *cluster* y el desarrollo de aplicaciones para utilizarlo.

El ancho de banda para la interconexión de las computadoras que conforman un *cluster* sigue siendo reducido en comparación con el que puede lograrse a través de un bus interno de una computadora paralela. Sin embargo, las tecnologías de redes están avanzando de forma tal que esta situación podría subsanarse en un futuro mediato.

Por otro lado, siempre es posible considerar la limitante del ancho de banda al momento de diseñar las aplicaciones, para minimizar la cantidad de información a transmitir entre los diferentes procesadores y evitar la saturación del medio de comunicación.

Existen varias opciones para la implementación del modelo de comunicación de paso de mensajes. Las librerías comúnmente usadas son PVM (*Parallel Virtual Machine*) y MPI (*Message Passing Interface*).

3.1.1. MPI

MPI es una estandarización de las librerías de paso de mensajes que fue creada por el MPI-FORUM, un grupo formado por investigadores de universidades, laboratorios y empresas involucradas en la computación de alta prestación. Se abocaron a estandarizar las principales funciones de las diferentes librerías que existían para el paso de mensajes y lograron presentar un primer estándar en noviembre de 1993 [16].

MPI, como estándar, es portable y originalmente tenía librerías para ser implementado en lenguage C, C++, y Fortran. Actualmente también hay librerías para Java, Phyton y Perl, entre otros.

Existen varias implementaciones del estándar; por ejemplo, empresas como Intel, IBM, Cray Research tienen sus propias versiones del estándar; también existen implementaciones de software libre como son: MPICH y OpenMPI.

Capítulo 4

Algoritmos existentes para TSP

4.1. Introducción

Siempre que hablamos de un problema con una cantidad finita de soluciones posibles y considerando la capacidad de procesamiento de los equipos de cómputo modernos y el uso de las estrategias de paralelización antes mencionadas, es natural preguntarse: ¿Podemos enumerar todas las soluciones y simplemente elegir la mejor? Este método es conocido con el nombre de fuerza bruta.

Dada una instancia de TSP simétrico con m nodos $n_1, n_2, ..., n_m$, resulta bastante intuitivo que para construir un recorrido (dado que debemos tomar todos ellos sin repetición) existen $m!$ recorridos posibles. Esto se ve fácilmente si consideramos que se tienen m nodos a eleguir para comenzar, $m-1$ posibles para la segunda ciudad del recorrido, $m-2$ para la siguiente y así sucesivamente hasta llegar a la última ciudad del recorrido, luego $(m)(m-1)(m-2)\cdots(1) = m!$.

Hay que considerar que el problema se puede simplificar un poco, ya que dada una ruta, nos da igual cualquier punto de partida y esto reduce el número de rutas a examinar en un factor m, lo que resulta en un espacio de búsqueda de tamaño $(m-1)!$. Como no importa la dirección en que se desplace el agente, el número de rutas a examinar se reduce nuevamente en un factor de 2. Por lo tanto hay que considerar $(m-1)/2$ rutas posibles.

En la práctica, para un problema con 5 ciudades hay 12 rutas diferentes y no necesitamos una computadora para encontrar la mejor ruta, pero apenas aumentamos el número de ciudades las

posibilidades crecen exponencialmente (en realidad, factorialmente): Para 10 ciudades hay 181,440 rutas diferentes, Para 30 ciudades hay más de 10^{31} rutas posibles. Una computadora que calcule un millón de rutas por segundo necesitaría 10^{18} años para resolverlo. Dicho de otra forma, si hubiera empezado a calcular todas las rutas desde el comienzo de la creación del universo (hace unos 13'400 millones de años) todavía no habría terminado.

Enfoques mucho más sofisticados han sido aplicados a TSP, siendo la estrategia basada en programación dinámica desarrollada por Richard Bellman y G.B. Dantzing [17], la que es capaz de garantizar la solución óptima en un menor tiempo (una complejidad de orden $O(n^2 2^n)$, que sigue siendo muy grande computacionalmente pero es claramente mejor al orden $O(n!)$ obtenido mediante fuerza bruta.

Debido a que TSP es un problema NP-Duro y no se conocen algoritmos eficientes (de tiempo polinomial) que hallen la solución exacta, nos enfocaremos en métodos eficientes que sólo pretenden encontrar soluciones razonablemente buenas. Existen dos categorías de tales algoritmos, los algoritmos de aproximación y las heurísticas.

El primer algoritmo de este tipo para atacar TSP en el que podemos pensar se conoce como algoritmo del vecino más próximo (NN por sus siglas en inglés); consiste en posicionarse en un nodo arbitrario y tomar la arista más corta que lo conecte con algún nodo no visitado, desplazarse a dicho nodo y repetir el proceso hasta que no queden más nodos por visitar, este método genera rápidamente un camino corto, pero generalmente no el ideal. Suele utilizarse para inicializar algoritmos más elaborados. Como norma general, si los últimos pasos del recorrido son de magnitud similar a los primeros, el recorrido es razonable; si estos son mucho mayores, es probable que el camino no sea óptimo.

Para N ciudades aleatoriamente distribuidas en un plano, el algoritmo produce en promedio un camino 25 % más largo que el óptimo posible. Sin embargo, existen muchos casos donde la distribución de las ciudades hace que el algoritmo NN devuelva el peor camino. [18].

4.2. Algoritmos de aproximación

Un algoritmo de aproximación es aquel que, a lo sumo, difiere del óptimo en un cierto porcentaje calculable y tiene un orden polinomial.

Sea A un algoritmo de tiempo polinomial aplicado a un problema NP-Duro que obtiene un valor $A(I)$ aproximado al óptimo para cada instancia I del problema y sea $\mathrm{Opt}(I)$ el valor óptimo. Estamos interesados en acotar el error relativo de $A(I)$.

Decimos que un problema de minimización es aproximable si existe $k \geqslant 1$ tal que para cualquier instancia I del problema: $A(I) \leqslant k{*}\mathrm{Opt}(I)$, o en caso de un problema de maximización $A(I) \geqslant k{*}\mathrm{Opt}(I)$ con $0 < k \leq 1$.

Veremos tres casos:

1. Se satisface la desigualdad para algún $k \geqslant 1$. En este caso decimos que el problema es k-aproximable.

2. Se satisface la desigualdad para cualquier $k \geqslant 1$ prefijado.

3. No se puede acotar el error de la forma antes descrita. En este caso, decimos que el problema es no-aproximable.

Un algoritmo de aproximación para el TSP necesita encontrar solamente una permutación de las ciudades cuyo valor $A(I)$ esté cerca de $\mathrm{Opt}(I)$.

Si $P \neq NP$ entonces el TSP no es aproximable en tiempo polinomial; si lo fuese eso implicaría que el problema del ciclo hamiltoniano es P. Esto se puede demostrar de la siguiente manera:

Sea G un grafo, supongamos que existe A de tiempo polinomial tal que $A(x) \leq$ k*Opt(x) para el TSP. Si definimos el costo entre los nodos (i, j) como:

$$c_{ij} = \begin{cases} 1 & \text{si}(i,j) \in E \\ 2 + n(k-1) & \text{si no} \end{cases}.$$

entonces tenemos un grafo completo con costos. Aplicamos A y pueden ocurrir sólo dos cosas:

1. Todas las aristas del ciclo tienen costo $= 1$. Entonces $A(x) = n$ en cuyo caso G tiene un ciclo hamiltoniano.

2. Alguna rama tiene costo $2 + n(k-1)$ y las restantes costo $\geqslant 1$. Por lo tanto $A(x) \geqslant n-1+2+n(k-1) = kn+1 > kn$, de lo que se deduce que G no puede tener un ciclo hamiltoniano (ya que si lo tuviera, entonces $\text{Opt}(x) = n \therefore kn = k \text{ Opt } (x)$).

Se deduce que no existe un algoritmo polinomial que aproxime a TSP en su forma general. Sin embargo, para la versión de TSP simétrico donde la distancia además sea una métrica (TSP triangular) el algoritmo de Christofides es un algoritmo de aproximación 3/2 [19] .

4.2.1. Algoritmo de Christofides

El algoritmo de Christofides, desarrollado en 1976 por Nicos Christofides profesor del Imperial College London [20], permite aproximar instancias de TSP en que los pesos de las aristas satisfacen la desigualdad triangular.

Sea $G = (V, w)$ una instancia de TSP, en donde G es un grafo completo definido por un conjunto de vértices V y una función w que asocia un costo a cada arista del grafo G.

A continunación se describen las fases del algoritmo en pseudocódigo:

1. Obtener el árbol recubridor mínimo T de G.

2. Sea O el conjunto de vértices de grado impar en T, hallar un recubrimiento perfecto M de mínimo peso en el grafo completo sobre los vértices de O.

3. Combinar las aristas de M y T para crear un grafo H.

4. Obtener un ciclo euleriano en H (H se considera euleriano si es conexo y solo presenta vértices de grado par).

5. Obtener un ciclo hamiltoniano a partir del ciclo euleriano anterior, descartando los nodos visitados.

El costo de la solución obtenida por el algoritmo es máximo 3/2 de la solución óptima.

La prueba es la siguiente:

Sea A el conjunto de aristas de la solución óptima del TSP para G. Como (V,A) es conexo, contendrá varios árboles recubridores T, y por tanto, $w(A) \geqslant w(T)$. Además, sea B el conjunto de

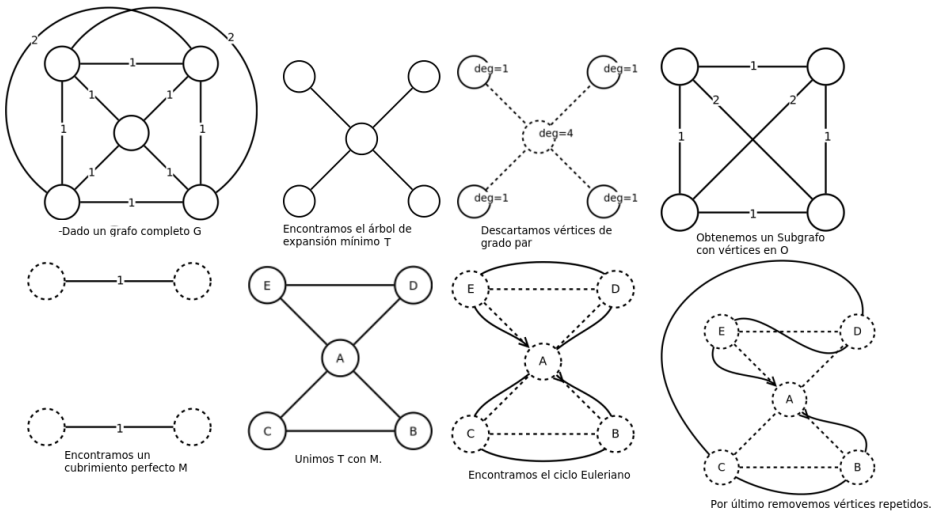

Figura 4.1: Ejemplo del algoritmo de Christofides

aristas de la solución óptima del TSP para el grafo completo sobre los vértices de O. Como los pesos asociados a las aristas son "triangulares"(visitar más nodos no reduce, en ningún caso, el coste total), se tiene que $w(A) \geqslant w(B)$. Se demuestra así que existe un cubrimiento perfecto de vértices de O con peso tal que $w(B)/2 \leq w(A)/2$, de forma que este límite constituye una cota superior para M (ya que M es un cubrimiento perfecto de mínimo coste). Debido a que O debe contener un número par de vértices, existe un cubrimiento perfecto. Sea $e_1, ..., e_{2k}$ un camino euleriano en (O,B). Es evidente que tanto $e_1, e_3, ..., e_{2k-1}$ como $e_2, e_4, ..., e_{2k}$ son cubrimientos perfectos, y que el peso de (al menos) uno de ellos es menor o igual que $w(B)/2$. Así, se tiene que $w(M)+w(T) \leqslant w(A)+w(A)/2$, y debido a que descartar aristas no aumenta el costo total se deduce que el algoritmo se aproxima en 3/2 al óptimo.

Un ejemplo del algoritmo de Christofides se puede apreciar en la Fig. 4.1

4.3. Programación lineal y cotas

La programación lineal o PL es una técnica cuantitativa utilizada por la investigación de operaciones, que pretende encontrar mediante el uso de funciones lineales una solución óptima a un problema indeterminado [21].

Consiste en optimizar una función lineal, denominada función objetivo, de tal forma que las variables de dicha función estén sujetas a una serie de restricciones que expresamos mediante un

sistema de inecuaciones lineales.

Para poder plantear TSP simétrico y triangular como un problema de programación lineal es importante hacer una serie de observaciones: la primera es que, para cualquier recorrido simétrico, todos los vértices deben intersectar exactamente dos aristas; luego, si definimos $X_{i,j}$ como en (4.1) y $C_{i,j}$ como el costo de ir del nodo i al j, el problema de minimización (4.2) incluye a todos los recorridos hamiltonianos para m ciudades. Pero no todas las soluciones válidas del problema de PL son un recorrido hamiltoniano, de ahí se desprende que la solución de TSP nunca puede ser menor que el óptimo del problema de PL (encontramos una cota para el costo).

$$X_{i,j} = \begin{cases} 0 & \text{si la arista que conecta } i \text{ con } j \text{ no está en el recorrido} \\ 1 & \text{en caso contrario} \end{cases} \tag{4.1}$$

$$\begin{aligned} \min\ C_{1,2}X_{1,2} + C_{1,3}X_{1,3} + ... + C_{m-1,m}X_{m-1,m} \\ \text{Sujeto a: } \sum_{i=1, i\neq k}^{m} X_{k,i} = 2 \forall i, k \leqslant m \\ X_{i,j} \geqslant 0 \end{aligned} \tag{4.2}$$

Otra cota se puede encontrar suponiendo que se dibuja un disco de radio r centrado en un nodo, de tal manera que dicho disco no toca otros nodos. Como el agente en algún punto debe visitar ese nodo y, para hacerlo debe recorrer una distancia r de ida y de vuelta, entonces el recorrido mínimo debe ser mayor que $2r$. Si encontramos un radio para cada ciudad, de tal forma que los circulos de todas las ciudades no se toquen, la suma del doble de todos los radios (llamados radios de control, Fig. 4.2) es una cota aún mejor para TSP. Esto puede plantearse como problema de PL (4.3).

$$\begin{aligned} \max\ 2r_1 + 2r_2 + ... + 2r_m \\ \text{Sujeto a: } r_i + r_j \leq C_{i,j} \end{aligned} \tag{4.3}$$

Estas cotas todavía pueden estar alejadas de la realidad, debido sobre todo a que en la práctica las ciudades se suelen encontrar en grupos, por ejemplo, considere el caso de la Fig. 4.3:

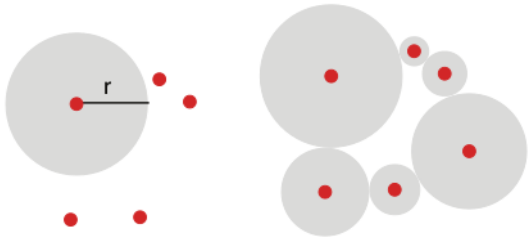

Figura 4.2: Radios de control

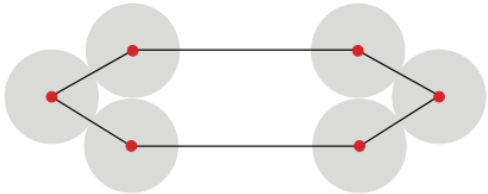

Figura 4.3: Peor caso, radios de control

Para solucionar estos problemas, se implementa una técnica conocida como *moat* (en Inglés), que consiste en agrupar nodos cercanos o vecinos en un conjunto de radios de control y agregar un espacio extra mientras no se crucen con otros grupos de nodos. Le sumamos al problema de PL la suma de dos veces la anchura mínima de todos los *moat*, a la que llaman *width*, y obtenemos una cota aún mejor, como se aprecia en Fig. 4.4.

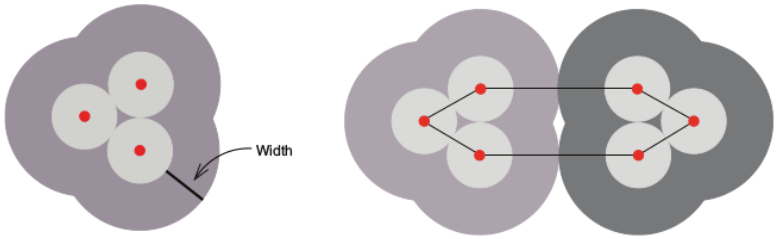

Figura 4.4: *Moat* para rellenar espacios

Se pueden agregar restricciones al problema de programación lineal para convertirlo en algo muy cercano a TSP, agregando reestricciones tales como que existan al menos dos vértices que unan cada par de grupos de ciudades, etc.

Se han encontrado, además, cotas superiores para el recorrido óptimo en función de las cotas inferiores encontradas con los métodos de PL con reestricciones. Sin embargo, no es posible modelar con precisión TSP como un problema de PL "puro" (de variables continuas). Es necesario utilizar técnicas de programación entera, como el método simplex con un árbol de búsqueda o alguna de

sus variantes.

El método simplex [21] se puede resumir en tres pasos:

1. Sea P el polítopo (polígono generalizado de n dimensiones) que satisface el conjunto de todas las reestricciones del problema de PL.

2. Encontrar un punto extremo o vértice en P desde el cual comenzar, una solución factible al problema.

3. Desplazarse desde ese punto extremo a otro, a lo largo de alguna arista de P (cada desplazamiento se conoce como pivoteo).

4. Repetir el paso 3 hasta que alcance la solución óptima o la arista escogida lleve hasta $-\infty$.

Pese a que el pivoteo consume un tiempo polinomial y el método funciona bastante bien para problemas pequeños, para problemas grandes, (por ejemplo una instancia de TSP en la que se busque asegurar que la solución sea un recorrido hamiltoniano) el polítopo resultante puede tener "tantos vértices como estrellas en el cielo" [6], debido a que cada restrición es un hiperplano.

No se sabe si existe alguna manera de elegir la solución inicial y una regla para hacer las mejoras que requiera sólo una cantidad polinomial de pivoteos. Al contrario, muchas reglas lógicas no son polinomiales.

En 1979 Leonid Jachián demostró, mediante un método al que llamó "Algoritmo de la elipsoide", que la PL podía ser computada siempre en tiempo polinomial [22], un descubrimiento sumamente importante en el área, aunque actualmente se considera poco práctico para instancias grandes, debido al alto grado polinómico de su complejidad computacional. Algunas variantes del método simplex han probado ser mucho más rápidas para instancias grandes [23].

4.4. Algoritmos heurísticos para el TSP

Una alternativa a las estrategias mencionadas anteriormente son los algoritmos heurísticos. Dichas técnicas consisten en procedimientos sistemáticos de prueba que ofrecen soluciones aceptables,

no necesariamente óptimos globales, para problemas donde el espacio de soluciones es indeterminado o lo suficientemente amplio como para que no pueda ser evaluado en un tiempo razonable. En muchos casos, las técnicas heurísticas se diseñan en función de las características particulares del problema a resolver [24].

Métodos modernos pueden encontrar soluciones para problema extremadamente largos (miles de ciudades) en un tiempo razonable [25].

Se puede establecer diferentes clasificaciones para las técnicas heurísticas en función de sus características especiales. Un aspecto diferenciador radica en el número de soluciones que se utilizan en el proceso de optimización [24]. Heurísticas como Recocido Simulado [26] o Búsqueda Tabú [27] utilizan una sola solución durante el proceso de búsqueda. Se suelen denominar métodos de trayectoria, ya que la solución describe una trayectoria desde la solución de partida hasta encontrar la solución final [24]. Por otro lado, técnicas como los Algoritmos Genéticos [28] o la Optimización por Colonia de Hormigas [29] hacen uso de un conjunto de soluciones (población) que son optimizadas de forma simultánea durante la búsqueda, como se observa en la Fig. 4.5 . Normalmente los métodos poblacionales suelen conseguir soluciones de más calidad debido a la ventaja que supone explorar simultáneamente diferentes áreas del espacio de búsqueda, pero al mismo tiempo suelen ser más costosos computacionalmente.

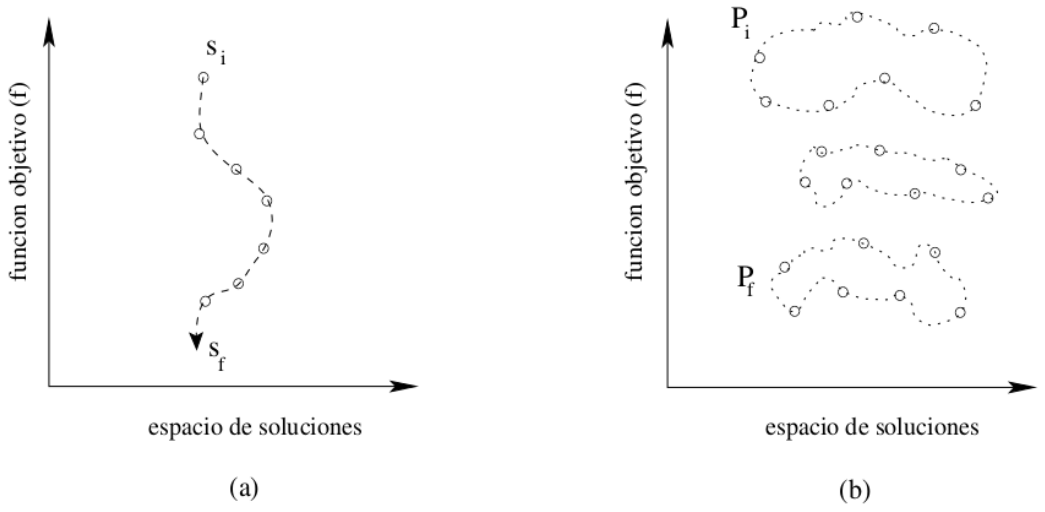

Figura 4.5: a. Heurística basada en trayectorias, b. poblacional.

Dentro de las técnicas basadas en trayectorias, existen algunas cuyo objetivo es describir una

trayectoria hacia un óptimo local y no la exploración exhaustiva del espacio de búsqueda, estos algoritmos se conocen como de búsqueda local o de mejoras iterativas.

4.4.1. Algoritmos de mejoras iterativas

Muchos problemas de optimización combinatoria pueden ser formulados en forma abstracta como "encontrar en un conjunto S un subconjunto T que satisface cierto criterio C y minimiza una función objetivo f".

Por ejemplo, en el TSP tenemos que encontrar en el conjunto de todas las ramas de un grafo completo, un subconjunto que forme un tour y tenga distancia mínima. Un enfoque heurístico para los problemas de optimización combinatoria es la mejora iterativa de un conjunto de soluciones factibles seleccionadas aleatoriamente.

La estructura de este tipo de algoritmo es:

1. Generar al azar una solución factible, es decir, un conjunto T que satisface C.

2. Tratar de encontrar una solución factible mejor T' por alguna transformación de T.

3. Si se encuentra una solución mejor, es decir, tal que $f(T') < f(T)$, entonces se reemplaza T por T'.

4. Si no se puede encontrar una solución mejor, T es una solución óptima local.

5. Repetir desde el paso 1 hasta que se termine el tiempo dejado de antemano o la respuesta sea satisfactoria.

El corazón de este procedimiento iterativo es el paso 2, en donde se intenta mejorar la solución dada. Una transformación que ha sido aplicada a una gran variedad de problemas es el cambio de un número fijo k de elementos de T con un número k de elementos en $S - T$, de forma que la solución resultante T' sea factible y mejor. Este mecanismo se repite hasta que no sea posible mejorar T a través de estos cambios [7]. El principal problema es encontrar los elementos correctos a reemplazar.

En 1956 Croes [30] propuso un algoritmo de búsqueda local para TSP, que actualmente se conoce como 2-opt cuya idea central es tomar un recorrido hamiltoniano y llevar a cabo el reemplazo de una pareja de aristas en el recorrido por una pareja de aristas que no se encuentren en el mismo, con base al hecho de que si la ruta tiene alguna intersección se puede reducir el costo simplemente invirtiendo el orden en que se visitan los nodos en el subrecorrido entre los nodos en que se produce la intersección, como se puede ver en la Fig. 4.6.

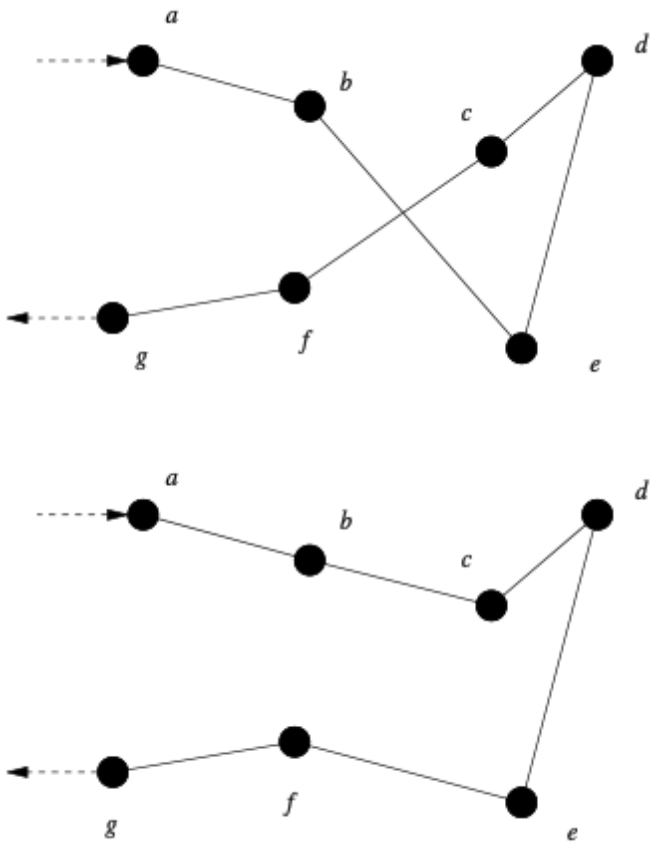

Figura 4.6: Intercambio de las aristas (b,e) y (f,c) por las aristas (b,c) y (f,e)

Estrategias de búsqueda local basadas en este algoritmo pero con k reemplazos, fueron discutidas por Bock en 1963 [31]. A todas las variantes de este algoritmo se les conoce como algoritmos k-opt.

Es importante mencionar que el número de reconexiones posibles después de quitar k aristas, tal que el recorrido resultante es hamiltoniano, crece de forma factorial. Dicho número es $(k-1)!2^{k-1}$ [48], esto aunado a que las formas distintas en que se pueden cortar k aristas crece también de forma factorial, hacen que se considere demasiado costoso implementar estrategias k-opt para $k > 4$ [31].

Se han desarrollado múltiples algoritmos de búsqueda local basados en estrategias k-opt. Uno muy importante es el desarrollado por Shen Lin y Brian Kernighan en 1973 [32]. Su heurística, basada en k-opt, decide de manera heurística en cada iteración cuántas aristas debe reemplazar. El algoritmo de Lin-Kernighan produce aproximaciones cercanas al óptimo para instancias de TSP de hasta 100 ciudades. Sin embargo, tiene el inconveniente de ser bastante más lento que sus "hermanos" 2-Opt y 3-Opt, además del hecho que su programación es bastante compleja y requiere mucho almacenamiento en memoria, lo cual vuelve bastante difícil su aplicación en arquitecturas paralelas.

4.4.2. Algoritmos poblacionales

Dentro de las heurísticas basadas en poblaciones se destacan los Algoritmos Evolutivos (EA) y los algoritmos basados en Inteligencia Colectiva. Ambos tienen en común el haber sido inspirados en procesos naturales. El primero en la teoría de la evolución de Darwin-Wallace [28] y el segundo en ejemplos biológicos de comportamiento colectivo, como es el caso de las colonias de hormigas, los enjambres de abejas, las bandadas de aves y los cardúmenes de peces.

El término EA engloba una serie de técnicas inspiradas biológicamente, para simular el proceso evolutivo en una computadora. En terminos generales se requiere: [33]

- Codificar las estructuras que se replicarán.
- Operaciones que afecten a los individuos.
- Una función de aptitud.
- Un mecanismo de selección.

En el enfoque de optimización, cada uno de los individuos es una solución candidata del problema a resolver. Aplicado a TSP, cada individuo es un conjunto de aristas que definen un recorrido; dado que se recorren todas las ciudades al menos una vez, los individuos pueden ser representados como una permutación que indica el orden en que se visitan las ciudades en la solución candidata.

4.4.3. Algoritmos genéticos

Los algoritmos genéticos (AG) propuestos por Holland son métodos heurísticos inspirados en, además de la teoría de Darwin-Wallace, las teorías de Mendel y Watson acerca de la recombinación genética y las mutaciones en los seres vivos. [28].

Los algoritmos genéticos inician con una población generada aleatoriamente, después se evalúan sus individuos y se aplican los operadores genéticos para producir la población de la generación siguiente. Este proceso se repite hasta que se cumple con alguna condición de parada, por ejemplo un número determinado de generaciones o ciclos.

La aptitud de los individuos nos indica qué tan bien satisface o soluciona un individuo el problema de optimización, y sirve como criterio para discriminar a diferentes individuos dentro de una población.

El paso en el que se evalúa la aptitud de los individuos suele ser el más costoso para una aplicación real; puede ser una subrutina, un simulador, o cualquier proceso externo. Muchas veces se usan funciones aproximadas para reducir el costo de evaluación. Cuando existen restricciones éstas se pueden introducir en el costo como penalizaciones. En el contexto de TSP, la aptitud de los individuos o soluciones es directamente la suma del costo de cada una de las aristas en el recorrido que caracteriza dicho individuo, es decir el costo total del recorrido.

4.4.3.1. Selección

Una vez calificados todos los individuos de una generación, el algoritmo debe seleccionar a los individuos mejor adaptados para resolver el problema, de esta forma se incrementa la probabilidad de tener individuos o soluciones “buenas” en el futuro.

Existen diferentes métodos para seleccionar a los individuos de la siguiente generación. Uno de ellos es la selección directa, que consiste simplemente en especificar que permanezcan en la población los mejores n individuos. Otra estrategia es la llamada “selección por torneo”, que consiste en tomar muestras de n individuos e insertar en la siguiente generación al mejor de esos n individuos.

4.4.3.2. Cruza

Mientras que durante la selección solamente pasamos individuos de una generación a otra, es en el cruzamiento donde se producen nuevas soluciones. El cruzamiento consiste en tomar dos individuos o padres y a partir de ellos, producir nuevos individuos. En su formulación original donde las soluciones son cadenas binarias, la cruza consiste en dividir a los individuos viejos en dos secciones y recombinarlos para formar a su descendencia, lo que implica generar un entero i entre uno y la longitud de la cadena binaria (llamado punto de cruzamiento) y a partir de esa posición recombinar, como se observa en la Fig. 4.7

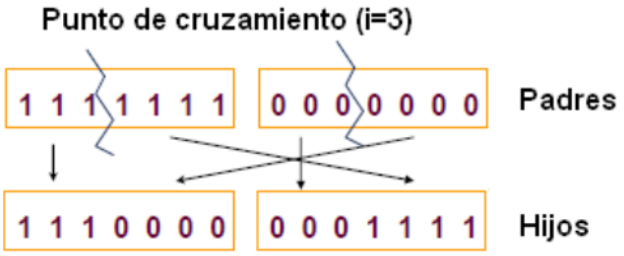

Figura 4.7: Cruza para cadenas binarias

Sin embargo, para la representación de permutaciones, al efectuar las técnicas tradicionales de cruza los hijos serán invariablemente no válidos [33]. Este problema requiere la definición de procedimientos de 'reparación' de cadenas inválidas que se producen a consecuencia de la cruza, uno de los procedimientos más simples para llevar a cabo dicha corrección se conoce como Order Crossover (OX) y consiste en el siguiente algoritmo (los padres son P1 y P2) [34].:

1. Seleccionar aleatoriamente una subcadena de P1.

2. Producir un hijo copiando la subcadena en las posiciones correspondientes a P1. Las posiciones restantes se dejan en blanco.

3. Borrar los valores que ya se encuentren en la subcadena de P2. La secuencia resultante contiene los valores faltantes.

4. Colocar los valores en posiciones no conocidas del hijo de izquierda a derecha.

5. Para obtener el segundo hijo, se repiten los pasos del 1 al 4, pero tomando ahora la sub-cadena de P2.

Otras estrategias de cruza para permutaciones son discutidas en [33].

4.4.3.3. Mutación

Ocasionalmente algunos individuos se alteran a propósito, los cuales se seleccionan aleatoriamente, en lo que constituye el símil de una mutación. El objetivo es generar nuevos individuos que exploren regiones del dominio del problema que probablemente no se han visitado aún.

Un método comunmente usado consiste en generar un valor aleatorio entre cero y uno, compararlo con la probabilidad de mutación (un valor previamente definido). En caso de ser menor se procede a alterar al individuo.

Bajo cadenas binarias, el operador de mutación suele implementarse tomando un bit aleatorio en la cadena e invertirlo (pasar de cero a uno o viceversa). Sin embargo, para generar individuos válidos bajo la representación de soluciones como permutaciones se utilizan otras estrategias como el intercambio de elementos en un par de posiciones aleatorias de la misma (intercambio recíproco).

En general, la probabilidad de que un individuo mute suele ser muy baja, para evitar que la búsqueda se vuelva demasiado aleatoria.

4.4.3.4. Características de un AG

Todo algoritmo de búsqueda debe establecer un equilibrio entre dos factores aparentemente contrapuestos:

- Exploración del espacio de soluciones para realizar una búsqueda en amplitud, localizando así zonas prometedoras, y
- Explotación del espacio de búsqueda, para hacer una búsqueda en profundidad en dichas zonas, obteniendo así las mejores soluciones.

Los algoritmos genéticos son un tipo de algoritmo de búsqueda de propósito general, cuyos operadores pueden establecer un equilibrio adecuado entre exploración y explotación. [35]

Las principales características que hacen a los algoritmos genéticos tan atractivos desde un punto de vista computacional y que los han hecho muy populares en muchos campos, además de dicho equilibrio, son:

- Las pocas condiciones que debe cumplir el espacio de búsqueda.
- La disponibilidad de varias metodologías probadas para su paralelización.

4.4.4. Algoritmos genéticos paralelos

La mayoría de los métodos de optimización, incluyendo las heurísticas tienen el inconveniente de converger en ocasiones a valores que parecen ser la solución del problema, pero que solo lo son para un subconjunto del espacio de búsqueda. Estas soluciones se conocen comon óptimos locales.

Una manera de evitar que un AG converja con frecuencia a óptimos locales es incrementar la probabilidad de mutación; otro enfoque son los algoritmos genéticos paralelos, en los cuales, al buscar simultaneamente en distintas partes del espacio de búsqueda, se minimizan las posibilidades de quedar atrapados en óptimos locales. E igualmente importante, al procesarse de manera paralela los cálculos de aptitud y las operaciones genéticas en distintas poblaciones, se reduce el tiempo de ejecución del algoritmo.

Nowostawski y Poli [36] proponen clasificar los algoritmos genéticos en 8 categorías, que obtuvieron considerando los siguientes rasgos:

- La forma de evaluar la aptitud y aplicar la mutación.
- Si se implementa en una sola población o varias sub-poblaciones.
- En el caso de que se implemente en varias sub-poblaciones, la forma en la que se intercambian individuos entre ellas.
- Si se hace selección global o localmente.

Las categorías propuestas son:

1. Maestro-Esclavo.

- Una sola población.
- Generalmente se parareliza el cálculo de aptitud y la mutación, a cargo de varios procesos esclavos.
- La selección y el cruce se llevan a cabo de manera global, a cargo de un proceso maestro.

2. Subpoblaciones estáticas con migración (modelo de islas).

   - Conjunto estático de subpoblaciones independientes.
   - Requiere implementar un operador genético adicional: migración, que consiste en intercambiar individuos entre subpoblaciones cada cierto número de generaciones. La migración permite compartir material genético entre las subpoblaciones y para implementarla requiere que se incluya en el algoritmo un conjunto de parámetros adicionales.
   - Los modelos de migración pueden ser:
     - En anillo: Antes de iniciar el algoritmo se determina la población destino de cada elemento a migrar.
     - Aleatoria: Se elige al azar la población destino, en el momento de ejecutar la migración.
   - Las estrategias para seleccionar los individuos que migrarán pueden ser:
     - Elitista: Se migran los mejores individuos de cada población.
     - Por torneo.

3. Subpoblaciones estáticas superpuestas.

   - Básicamente funcionan como la categoría anterior, solo que se disminuye la cantidad de individuos en la subpoblación.
   - Se implementan en computadoras masivamente paralelas.

4. AG masivamente paralelos

   - Básicamente funcionan como la categoría anterior, solo que se disminuye la cantidad de individuos en la subpoblación.

- Se implementan en computadoras masivamente paralelas.

5. Subpoblaciones dinámicas.

   - Una sola población.
   - Se divide en subpoblaciones en tiempo de ejecución.

6. AG de estado estable.

   - Se distinguen de los algoritmos maestro-esclavo por la forma en la que aplican la selección.
   - El algoritmo no espera a que una fracción de la población haya concluido su evaluación, sino que continúa con la población existente.

7. AG desordenados.

   - Constan de tres fases:
     - Inicialización: Se crea la población inicial, usando algún método de ennumeraicón parcial.
     - Primaria: Se reduce la población mediante alguna forma de selección y se evalúan sus individuos.
     - Yuxtaposición: Se unen las soluciones parciales resultantes de las fases anteriores.

8. Métodos híbridos.

   - Combinan características de los métodos anteriores.

Cada una de las categorías puede presentar mejores rendimientos, dependiendo del tipo de problema a tratar y de la arquitectura de la máquina paralela en la que se implementará la solución.

4.4.5. Algoritmos meméticos

Durante la segunda mitad del siglo XX se desarrolló una tendencia hacia el purísmo algorítmico, es decir a no apreciar la flexibilidad o capacidad de un algoritmo heurístico de asimilar elementos

externos pertenecientes al problema en particular que se buscase resolver. No fue hasta mediados de los noventa cuando la formulación del llamado teorema de "*No Free Lunch*" por Wolpert y Macready [37] dejó definitivamente claro que un algoritmo de optimización se comporta en estricta concordancia con la cantidad y calidad del conocimiento específico del problema que incorpora. Esta filosofía, que comenzó a imponerse de manera generalizada a finales del siglo XX, ya estaba siendo promulgada con anterioridad por diversos investigadores, e.g., Hart and Belew [38], Davis [39], y Moscato [40].

Los Algoritmos Meméticos (MA) surgirirían precisamente a partir del trabajo de Pablo Moscato [41, 42] y constituyen un paradigma de optimizacion basado en la explotación sistemática de conocimiento acerca del problema que se desea resolver, y de la combinación de ideas tomadas de diferentes heurísticas, tanto basadas en población como basadas en búsqueda local [43].

El adjetivo "memético" viene del término en inglés "*meme*" utilizado por R. Dawkins [44] para designar al análogo del gen en el contexto de la evolución cultural. Sin embargo, resulta conveniente resaltar que el empleo de esta terminología no representa un propósito de adherirse a una metáfora de funcionamiento concreta (la evolución cultural en este caso), sino más bien lo contrario: hacer explícito que se difumina la inspiración puramente biológica, y se opta por modelos más genéricos en los que se manipula, se aprende y se transmite información.

Pueden encontrarse diversos trabajos que hacen uso de nombres alternativos para referirse a estos (e.g., EAs híbridos o lamarckianos), o que aún usando el propio término MA, hacen una interpretación muy restringida del mismo.

Debido a la mezcla de las propiedades incluyentes de los algoritmos poblacionales y la exploración a detalle de zonas prometedoras de la búsqueda local, los MA han mostrado ser más eficientes y eficaces en la resolución de problemas combinatorios [3, 45, 46]

El pseudocódigo general de un MA basado en un AG se puede observar en el Algoritmo 1.

Algoritmo 1 Estructura general de un algoritmo memético

```
t=0
P(t)=InicializarLaPoblación
P(t)=BúsquedaLocal(P(t))
P(t)=Selección(P(t))
mientras criterio de parada no cumplido haz
    P(t)=Cruza(P(t))
    P(t)=Mutación(P(t))
    P(t)=BusquedaLocal(P(t))
    P(t+1)=Seleccion(P(t))
    t=t+1
fin mientras
```

Capítulo 5

Estrategia propuesta

Uno de los principales inconvenientes al trabajar sobre un *cluster* de computadoras es lo limitado del ancho de banda de interconexión entre procesadores, lo que da lugar al fenómeno conocido como "cuello de botella" que limita la comunicación entre procesadores. Este fenómeno ocurre cuando se realizan demasiadas solicitudes a la red, que al no poder ser atender todas ellas al mismo tiempo, puede producir colisiones, perdida de información o saturación.

El algoritmo genético distribuido con migración es particularmente adecuado para trabajar sobre la arquitectura del *cluster* debido a que los procesos son independientes unos de los otros excepto al llevar a cabo migración, la cual podemos controlar con los parámetros: periodo y tasa de migración. El primero indica cada cuántas generaciones se compartirán individuos entre procesos y el segundo el número de individuos que se compartirán. Al controlar dichos parámetros podemos reducir el uso del ancho de banda de comunicación entre procesos (con periodo de migración grande y tasa de migración pequeña).

En la Fig 5.1, podemos observar el diagrama de flujo general de un algoritmo genético con migración.

Aquí se propone utilizar un *cluster* de computadoras para implementar un algoritmo que se puede dividir en tres fases: La primera es la inicialización de los procesos (uno maestro y los restantes esclavos), la segunda fase es el núcleo o la optimización (que es un algoritmo memético basado en un algoritmo genético de subpoblaciones estáticas con migración) y una última fase en que cada proceso esclavo envía su mejor resultado al proceso maestro, para que encuentre el mejor

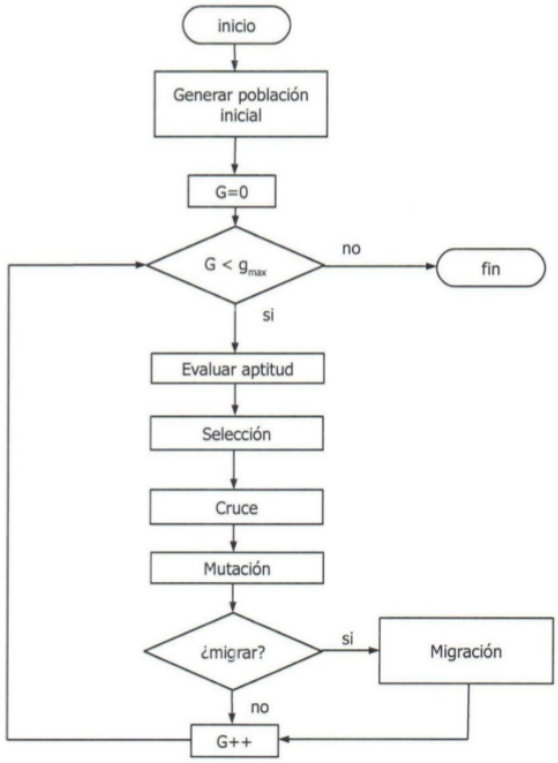

Figura 5.1: Diagrama de flujo de un AG con migración

de todos.

Para garantizar una alta diversidad en las soluciones iniciales de nuestro algoritmo memético, empezamos generando una población inicial de soluciones de forma aleatoria. Sobre dichas soluciones se aplica el algoritmo de búsqueda local.

Los procesos esclavos actúan sobre cada una de las subpoblaciones. Una vez que se cumple un cierto número de generaciones en el algoritmo memético, cada nodo esclavo comparte información con los demás en un símil a la migración. Esta estrategia conocida también como modelo de islas, ha demostrado producir mejores soluciones que poblaciones aisladas o una sola población masiva [47].

Cada vez que se generan nuevos individuos en la población, es decir, justo después de aplicar los operadores genéticos de cruza y mutación, se aplica el algoritmo de búsqueda local sobre dichos individuos, de tal forma que el algoritmo memético que se ejecuta en cada uno de los procesos esclavos es como el que se observa en el diagrama de flujo de la Fig. 5.2

La manera de llevar a cabo los operadores genéticos se decidió empíricamente mediante pruebas preliminares. Finalmente se utilizó una selección directa y una estrategia de cruce tipo OX.

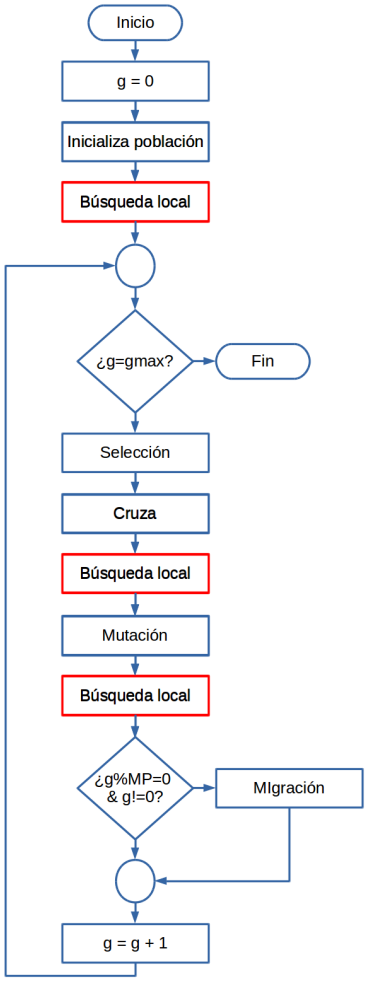

Figura 5.2: Diagrama de flujo del algoritmo memético propuesto

Las estrategias para llevar a cabo mutación bajo la representación de permutaciones más conocidas son: intercambio recíproco, por inserción o por desplazamiento simple. Todas ellas llevan a soluciones que se encuentran dentro de la vecindad en que se hace la búsqueda local en nuestro algoritmo, por lo que fue necesario implementar un operador de mutación distinto que nos permitiera aumentar la diversidad de nuestra búsqueda. Se optó por un operador conocido como "*double bridge*" [48], que consiste en reemplazar cuatro aristas no adyacentes en el recorrido original por cuatro aristas que no se encuentren en el mismo y conserven la propiedad de ser un recorrido hamiltoniano, como se muestra en la Fig. 5.3

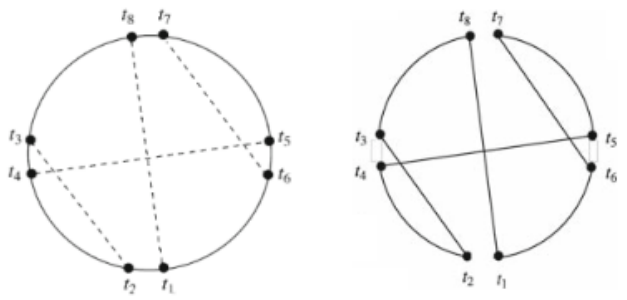

Figura 5.3: Ejemplo de un cambio *double bridge*

5.1. Algoritmo de búsqueda local propuesto

Debido a que las estrategias tradicionales k-opt que consideran todos los cambios posibles para $k > 2$ mejoran las soluciones de manera muy lenta, que 2-opt no lleva a soluciones demasiado cercanas al óptimo y que la más utilizada, Lin-Kernighan (LK), basada en determinar cuántas aristas cambiar en cada iteración conlleva un costo computacional demasiado alto, además requiere mucho almacenamiento en memoria y resulta difícil de programar, se propone un algoritmo de mejoramiento de rutas (búsqueda local) basado en realizar intercambios de tres aristas y en la estrategia LK.

De manera general, se realiza un intercambio de tres aristas x_1, x_2 y x_3 en el recorrido original T por tres aristas y_1, y_2 y y_3 que no se encuentren en T, como se aprecia en la Fig. 5.4, de tal forma que el costo total del nuevo recorrido sea menor.

La manera de seleccionar las primeras tres aristas (x_1, y_1, x_2) es como en la estrategia LK.

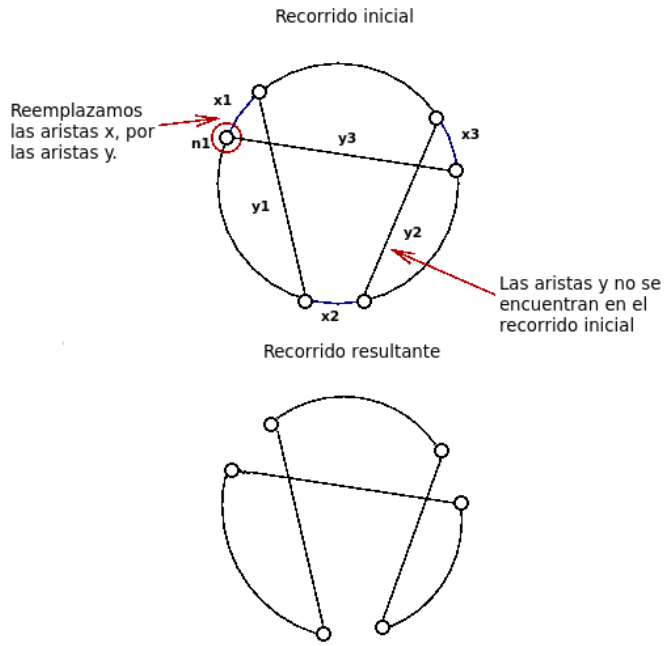

Figura 5.4: Ejemplo de iteración posible con reemplaza de tres aristas

Sin embargo, en lugar de buscar una arista y_2 que nos permita encontrar una arista x_3 como en dicha estrategia, realizamos una búsqueda de todas las combinaciones posibles (x_1, y_1, x_2, y_2, x_3, y_3) que contienen las aristas (x_1, y_1, x_2) seleccionadas anteriormente y producen un tour válido, probando primeramente cada valor posible x_3 y seleccionando (y_2, y_3) de tal forma que el intercambio de los elementos en $\{x_i\}$ por los elementos $\{y_i\}$ resulte en recorrido hamiltoniano con ganancia $g = (||x_1|| + ||x_2|| + ||x_2||) - (||y1_1|| + ||y_2|| + ||y_3||)$ positiva, en caso de no existir dicho conjunto se prueba con valores distintos para el primer conjunto de aristas, pero siempre con las mismas reestricciones que en LK, es decir y_1 debe compartir un nodo con x_1 y el otro con x_2, ser de costo menor que x_1 y no pertenecer al conjunto original de aristas de la solución.

Al seleccionar los valores de las aristas (x_1, y_1 y x_2) de dicha forma, reducimos el espacio de búsqueda original sustancialmente y con ello el tiempo de ejecución es mucho menor que si buscáramos todos los intercambios posibles de tres aristas. Posteriormente realizamos una búsqueda en profundidad sobre dicho espacio reducido y prometedor al considerar todas las aristas restantes x_3 en T, lo cual nos garantiza llegar a la mejor solución de dicho espacio de búsqueda reducido.

De manera detallada el algoritmo es el siguiente:

1. Se elige al azar un nodo n_1.

2. Se selecciona una arista $x_1 = (n_1, n_2)$ tal que $x_1 \in T$, donde T es el conjunto de aristas de la solución original, dando prioridad siempre a la de costo mayor de las dos aristas posibles.

3. Se elige una arista $y_1 = (n_2, n_3)$ que comparta un nodo con x_1 tal que $y_1 \notin T$ y $||y_1|| < ||x_1||$. Si no es posible encontrar dicha y_1, se regresa a 2 y se selecciona la otra arista x_1 posible. Si en dicho caso tampoco se puede encontrar y_1, volvemos a 1 y se selecciona otro nodo n_1.

4. Se selecciona una arista $x_2 = (n_3, n_4)$ que comparta un nodo con y_1 tal que $x_2 \in T$. Dado que existen dos aristas que satisfacen dichas condiciones se hace una selección tipo ruleta, asignando una mayor probabilidad de ser seleccionada a la arista de costo mayor.

5. Se elige de manera aleatoria $x_3 = (n_5, n_6)$, $x_3 \in T$, tal que $x_3 \neq x_1$, $x_3 \neq x_2$.

6. Se eligen y_2, y_3 tales que, al reemplazar (x_1,x_2,x_3) por (y_1,y_2,y_3), el recorrido resultante sea hamiltoniano (dichas aristas y_2,y_3 no son únicas).

7. Si algún intercambio encontrado en 6 produce un costo menor al inicial, se efectúa el reemplazo y se regresa al paso 1.

8. En caso contrario, si ningun par y_2, y_3 produce un recorrido mejor, regresamos a 5 y seleccionamos otra x_3.

9. En caso de haber probado todos los nodos x_3 posibles, se regresa al paso 3 seleccionando una arista y_1 distinta. Si ya se probó con todas las aristas y_1, entonces se ha encontrado un óptimo local y el algoritmo termina.

Para acelerar la velocidad del algoritmo de búsqueda local, se decidió llevar a los individuos primeramente a un óptimo relativo a cambios de dos aristas, con la estrategia 2-opt de Croes mencionada con anterioridad, para posteriormente aplicar a dichas soluciones el algoritmo 3-opt propuesto.

Capítulo 6

Pruebas y Resultados

Para el desarrollo del algoritmo memético se utilizó lenguaje C, compilado con GCC 4.8.4, un conjunto libre de compiladores distribuidos bajo la licencia GPL. Se utilizó la librería OpenMPI en su versión 1.6.4 para el paso de mensajes entre los procesos, sobre Linux Mint 17.3 de 64 bits.

Las pruebas fueron llevadas a cabo en un *cluster* de 4 computadoras con procesadores i7-3770 con 8 GB de memoria RAM para problemas de tamaño menor a 200 ciudades. Para problemas más grandes se utilizaron 8 computadoras con las mismas características.

Como parte del proyecto se desarrollaron los siguientes archivos que se incluyen como anexos a este trabajo:

- viajero.c (Código fuente principal).
- params.h (Archivo de encabezado donde se definen los parámetros del algoritmo).
- busqlocal.h (Archivo de encabezado donde se encuentran las funciones encargadas de llevar a cabo la búsqueda local).

Así como 20 instancias de tamaño mediano (entre 50 y 439 ciudades) de TSP para llevar a cabo las pruebas. Dichas instancias se obtuvieron de la librería TSPLIB: ch130, ch150, kroA100, kroA150, kroA200, kroB100, kroB150, kroB200, kroC100, kroE100, pr107, pr124, pr136, pr144, pr152, pr226, pr264, pr299, pr439, tsp225.

Primeramente se comparó el rendimiento del algoritmo de búsqueda local propuesto contra una versión acelerada de 2-opt y contra la forma tradicional del algoritmo 3-opt que verifica todos los

cambios posibles, al utilizarlos para aproximar soluciones de la instancia de 225 ciudades propuestas por Reinelt de la TSPLIB (tsp225), cuyo recorrido óptimo tiene una distancia de 3919.

Se generaron soluciones aleatorias y se operaron con dichos algoritmos. Los resultados se pueden apreciar en la tabla 6.1.

Tabla 6.1: Comparación de algoritmos de búsqueda local

Algoritmo	Distancia	Varianza	Tiempo	ERP
2-opt	4312.22	69.21	0.21s	10.21 %
3-opt	4053.39	33.57	8.27s	3.42 %
Propuesta	4044.68	45.64	0.60s	3.20 %

Posteriormente se decidió hacer la comparación del algoritmo memético desarrollado contra las estrategias heurísticas más utilizadas para resolver este problema: la optimización por colonia de hormigas (concebida originalmente para resolver TSP) y un AG distribuido (el más utilizado de todos los algoritmos heurísticos de optimización). Debido a la naturaleza estocástica de los algoritmos heurísticos, se decidió utilizar una muestra de 100 ejecuciones con cada una de las estrategias heurísticas.

En la tabla 6.2 se muestra una comparación entre un sistema de hormigas (AS), un algoritmo genético (AG) y el algoritmo memético propuesto (AM) al resolver la instancia de 100 ciudades propuestas por Krolak/Felts/Nelson de la TSPLIB. Dicha instancia tiene un tour óptimo con un costo de 21282. Las tablas 6.3, 6.4 y 6.5 muestran los parámetros para cada heurística.

Tabla 6.2: Comparación de Heurísticas

Algoritmo	Distancia	Tiempo	ERP
AS	25049	21.26s	17.7 %
AG	21925	13.69s	3.02 %
memético	21282	0.38s	0.0 %

Las tablas 6.6, 6.7 y 6.8 muestran los resultados obtenidos con el algoritmo propuesto para diferentes instancias del problema tomadas de TSPLIB. Los único parámetros que se modifican en cada caso son el tamaño de la población, el número de generaciones y de subpoblaciones, los cuales se muestran en cada tabla.

Tabla 6.3: Parámetros del Sistema de Hormigas Paralelo

Parámetro	Valor
Colonias	32
Iteraciones	100
Tamaño de la Colonia	1000
Factor de evaporación	0.1

Tabla 6.4: Parámetros del AG Distribuido

Parámetro	Valor
Subpoblaciones	32
Generaciones	5000
Población	1024
Probabilidad de Selección	50 %
Probabilidad de Mutación	20 %
Elitismo	10
Período Migratorio	50
Tasa de Migración	5

Tabla 6.5: Parámetros del algoritmo memético

Parámetro	Valor
Subpoblaciones	32
Generaciones	20
Población	10
Probabilidad de Selección	40 %
Probabilidad de Mutación	1 %
Elitismo	1
Período Migratorio	3
Tasa de Migración	1

Tabla 6.6: Resultados experimentales instancias Churritz y Reinelt (TSP225).

Instancia	Óptimo	Población	Generaciones	Subpoblaciones	ERP	Tiempo
ch130	6110	13	20	32	0.0 %	1.22s
ch150	6528	15	20	32	0.0 %	1.36s
tsp225	3919	22	20	64	0.0 %	8.20s

Tabla 6.7: Resultados experimentales instancias Krolak/Felts/Nelson.

Instancia	Óptimo	Población	Generaciones	Subpoblaciones	ERP	Tiempo
kroA100	21282	10	20	32	0.0 %	0.38s
kroB100	22141	10	20	32	0.0 %	0.38s
kroC100	20749	10	20	32	0.0 %	0.37s
kroD100	21294	10	20	32	0.0 %	0.43s
kroE100	22068	10	20	32	0.0 %	0.39s
kroA150	26524	15	20	32	0.0 %	1.53s
kroB150	26130	15	20	32	0.0 %	1.63s
kroA200	29368	20	20	64	0.0 %	6.87s
kroB200	29437	20	20	64	0.0 %	6.64s

Tabla 6.8: Resultados experimentales instancias Padberg/Rinaldi.

Instancia	Óptimo	Población	Generaciones	Subpoblaciones	Error promedio	ERP	Tiempo
pr107	44303	10	20	32	0	0.0 %	0.88s
pr124	59030	12	20	32	0	0.0 %	1.02s
pr136	96772	13	20	32	0	0.0 %	1.15s
pr144	58537	14	20	32	0	0.0 %	2.37s
pr152	73682	15	20	32	0	0.0 %	2.94s
pr226	80369	22	20	64	0	0.0 %	14.8s
pr264	49135	26	20	64	0	0.0 %	35.2s
pr299	48191	30	20	64	0	0.0 %	30.8s
pr439	107217	44	40	64	0.637	$6e^{-4}$ %	256s

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

Se desarrolló un algoritmo memético para TSP sacando provecho de las ventajas de los AG y la búsqueda local, se implementó el algoritmo en lenguaje de programación C, en una arquitectura paralela de bajo costo (un *cluster* de computadoras pertenenciente a la división de estudios de posgrado e investigación del ITLP).

Además, se desarrollaron cabeceras que permiten utilizar dicha implementación para resolver distintas instancias del problema, de tal manera que para resolver una instancia nueva basta con agregar una cabecera con las posiciones de los nodos y no hace falta modificar el código fuente del programa. Se añadieron cabeceras de hasta 20 instancias de TSP.

De acuerdo a los resultados obtenidos, el algoritmo de búsqueda local propuesto muestra ser una mejor alternativa que las búsquedas originales 2-opt y 3-opt, como se puede observar en la tabla 6.1, donde también se aprecia que la media del costo total del recorrido obtenido utilizando la estrategia propuesta es claramente mejor que la obtenida utilizando 2-opt, e incluso mejor que la obtenida utilizando 3-opt. Si bien en el caso de 3-opt la diferencia no es tan grande, el algoritmo propuesto es hasta diez veces más rápido y por tanto, más adecuado para implementarse en un algoritmo evolutivo poblacional.

El algoritmo memético basado en la estrategia de búsqueda local propuesta encuentra el recorrido óptimo de la instancia de 100 ciudades de Krolak, Felts y Nelson en un tiempo menor a

medio segundo. Por otro lado, utilizando un sistema de hormigas paralelo o un algoritmo genético distribuido no se alcanza el óptimo aún en tiempos cuarenta veces mayores que el requerido por el algoritmo memético de acuerdo a la tabla 6.2.

En las tablas 6.6, 6.7 y 6.8 se puede apreciar que el algoritmo memético encuentra el recorrido óptimo de las instancias utilizadas como prueba de hasta 300 ciudades. Para la instancia de 439 ciudades de Padberg/Rinaldi no se alcanza el óptimo en todas las ejecuciones, sin embargo el error promedio es muy pequeño.

7.2. Trabajo futuro

Los resultados conseguidos son más eficientes (mejores en tiempo y en calidad de los resultados) que los que se han obtenido utilizando estrategias heurísticas puras sobre *clusters* de computadoras, e incluso mejores que los obtenidos en arquitecturas de paralelización distintas como son los GPU's. Sin embargo, sería muy interesante implementar un algoritmo memético en estos últimos, dado que recientemente han obtenido resultados mejores, en relación a los *cluster*s, para otra clase de problemas.

Se intentó implementar la misma estrategia sobre una GPU bajo la arquitectura CUDA. Sin embargo los resultados obtenidos no fueron prometedores, debido a que el algoritmo de búsqueda local resultaba demasiado costoso para las pequeñas unidades de procesamiento paralelo (hilos) de la arquitectura, por lo que actualmente nos encontramos trabajando en el diseño de una estrategia distinta que resulte viable sobre GPU's.

Si bien los resultados ya son prometedores, también queda para futuras investigaciones probar esta estrategia con una cantidad mayor de instancias del problema para obtener conclusiones definitivas. Otra cuestión pendiente para trabajar en un futuro sería probar el algoritmo memético en problemas que modelen situaciones de la vida real.

También queda como trabajo futuro la comparación del algoritmo memético contra estrategias más elaboradas presentes en la literatura como la de Lin-Kernighan-Helsgaun, entre otras.

Capítulo 8

Bibliografía

[1] Z. Michalewicz and D. B. Fogel, *How to solve it: modern heuristics.* Charlotte, NC: Springer, 2 ed., 2000.

[2] A. Rosete, *Una Solución Flexible y Eficiente para el Trazado de Grafos Basada en el Escalador de Colinas Estocástico.* Tesis de doctorado, Instituto Superior Politécnico "José Antonio Echeverría", 200.

[3] P. Moscato and C. Cotta, "A Modern Introduction to Memetic Algorithms," in *Handbook of Metaheuristics*, vol. 146, pp. 141–183, 2010.

[4] M. A. Castro Fabela, *Algoritmos genéticos con edades para resolver el problema del viajero.* Tesis de maestría, Universidad Autónoma de Nuevo León, 1999.

[5] D. H. Abeledo, "Optimización Combinatoria y Programación Entera (http://www-2.dc.uba.ar/materias/ocom/)."

[6] W. Cook, *In Pursuit of the Traveling Salesman.* Princeton, New Jersey: Princeton University Press, 2011.

[7] L. Stockdale and D. S. Puddu, *El problema del viajante: un algoritmo heurístico y una aplicación.* Tesis de licenciatura, Universidad de Buenos Aires, 2011.

[8] T. Bektas, "The multiple traveling salesman problem: An overview of formulations and solution procedures," 2006.

[9] F. Meunier and P. Sarrabezolles, "Multicolor traveling salesman problem: approximation and feasibility," tech. rep., Université Paris Est, CERMICS, Marne-la-Vallée, France, 2013.

[10] G. Reinelt, "TSPLIB– A Traveling Salesman Problem Library," *ORSA Journal on Computing*, vol. 3, no. 4, p. 376, 1991.

[11] The Clay Mathematics Institute, "The Millenium Prize Problems," *Centre Belge de Recherches Mathematiques*, p. 0, 2014.

[12] R. M. Karp, "Reducibility among combinatorial problems," in *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*, pp. 219–241, 2010.

[13] J. A. Castro, M. A. Castro Liera, and I. Castro Liera, *Programación Paralela Aplicada en Optimización*. La Paz, B.C.S.: Instituto Tecnológico de La Paz, 1 ed., 2012.

[14] NVidia, *NVIDIA CUDA C Programming Guide*. No. 350, 2014.

[15] J. L. Hennessy, D. A. Patterson, and J. R. Larus, *Computer organization and design : the hardware/software interface*. San Francisco: Kaufmann, 2 ed., 1999.

[16] R. L. Graham, T. S. Woodall, and J. M. Squyres, "Open MPI: A Flexible High Performance MPI," *Proceedings 6th Annual International Conference on Parallel Processing and Applied Mathematics*, pp. 228–239, 2005.

[17] G. B. Dantzig, D. R. Fulkerson, S. M. Johnson, V. Chvatal, and W. J. Cook, "Solution of a large-scale traveling-salesman problem," in *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*, pp. 7–28, 2010.

[18] D. S. Johnson and L. A. McGeoch, *The Traveling Salesman Problem and its Variations*, vol. 12. Kluwer Academic Publishers, 2002.

[19] G. Cornuejols and G. L. Nemhauser, "Tight bounds for christofides' traveling salesman heuristic," *Mathematical Programming*, vol. 14, no. 1, pp. 116–121, 1978.

[20] N. Christofides, "Worst-Case Analysis of a New Heuristic for the travelling salesman problem," Tech. Rep. February, 1976.

[21] G. B. Dantzig, "Linear Programming," *Operations Research*, vol. 50, no. 1, pp. 42–47, 2002.

[22] L. G. Hačijan, "A polynomial algorithm in linear programming," *Dokl. Akad. Nauk SSSR*, vol. 244, no. 5, pp. 1093–1096, 1979.

[23] F. Walters, "Sequential simplex optimization," *Analytical Letters*, vol. 32, no. 2, p. 193, 1999.

[24] R. B. n. Navarro, *Meta-Heurísticas híbridas para optimización mono-objetivo y multi-objetivo. Paralelización y aplicaciones.* 2008.

[25] E. Elbeltagi, T. Hegazy, and D. Grierson, "Comparison among five evolutionary-based optimization algorithms," *Advanced Engineering Informatics*, vol. 19, no. 1, pp. 43–53, 2005.

[26] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing.," *Science (New York, N.Y.)*, vol. 220, no. 4598, pp. 671–80, 1983.

[27] F. Glover and M. Laguna, "Tabu Search," *Journal of computational biology a journal of computational molecular cell biology*, vol. 16, no. 12, pp. 1689–703, 1997.

[28] J. H. Holland, *Adaptation in Natural and Artificial Systems: An introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* 1975.

[29] M. Dorigo and L. M. Gambardella, "Ant colonies for the travelling salesman problem," *Biosystems*, vol. 43, no. 2, pp. 73–81, 1997.

[30] G. A. Croes, "A Method for Solving Traveling-Saleman Problems," *Operations Research*, vol. 6, no. 6, pp. 791–812, 1958.

[31] F. Bock, *"Mathematical programming solution of traveling salesman examples", in: Recent Advances in Mathematical Programming.* 1963.

[32] S. Lin and B. W. Kernighan, "An Effective Heuristic for the Traveling Salesman Problem," *Operations Research*, vol. 21, pp. 498–516, 1973.

[33] C. A. Coello Coello, "Introducción a la Computación Evolutiva," tech. rep., CINVESTAV-IPN, México D.F., 2011.

[34] L. Davis, "Job shop scheduling with genetic algorithms.," in *Proceedings of the 1st International Conference on Genetic Algorithms, 136-140.*, (NJ, USA), pp. 136–140, L. Erlbaum Associates Inc., jul 1985.

[35] U. Granada, "Soft Computing and Intelligent Information Systems.," 2008.

[36] M. Nowostawski and R. Poli, "Parallel genetic algorithm taxonomy," *International Conference on Knowledge-Based Intelligent Information Engineering Systems*, pp. 88–92, 1999.

[37] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[38] W. Hart and R. Belew, "Optimizing an arbitrary function is hard for the genetic algorithm," in *4th International Conference on Genetic Algorithms*, pp. 190–195, 1991.

[39] L. Davis, "Handbook of Genetic Algorithms," *Artificial Intelligence 100*, pp. 1–6, 1991.

[40] P. Moscato, "On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms.," tech. rep., California Institute of Technology, Pasadena California, USA, 1989.

[41] P. Moscato and M. G. Norman, "A memetic approach for the traveling salesman problem implementation of a computational ecology for combinatorial optimization on message-passing systems," *Parallel Computing and Transputer Applications*, vol. 28, no. 1, pp. 177–186, 1992.

[42] P. Moscato, "Memetic Algorithms," in *Handbook of Applied Optimization*, pp. 157–167, 2002.

[43] C. Cotta, "Una visión general de los algoritmos meméticos," tech. rep., Dept. Lenguajes y Ciencias de la Computación, ETSI Informática, University of Málaga, Campus de Teatinos, 29071, Málaga, Espana, 2007.

[44] R. Dawkins, "The Selfish Gene," *30th Anniversary Edition–with a new Introduction by the Author*, p. 384, 1976.

[45] W. Hart, N. Krasnogor, and J. Smith, "Memetic evolutionary algorithms," *Recent advances in memetic algorithms*, pp. 3–27, 2005.

[46] P. Merz and B. Freisleben, "Memetic Algorithms for the Traveling Salesman Problem," *Complex Systems*, vol. 13, pp. 297–345, 1997.

[47] M. A. Castro Liera, *Un algoritmo genético distribuido con aplicación en la identificación difusa de un proceso fermentativo.* Tesis de doctorado, Universidad Central, "Marta Abreu"de Las Villas, 2009.

[48] A. Blazinkas and A. Misevicius, "Combining 2-OPT, 3-OPT and 4-OPT with K-SWAP-KICK perturbations for the traveling salesman problem," tech. rep., Kaunas, Lithuania, 2009.

Anexo: Código fuente

8.1. Código principal (viajero.c)

```
/*
        viajero.c (codigo principal)

        Copyrigth (c) 2016 Joel Artemio Morales Viscaya : iscviscaya@gmail.
        Marco Antonio Castro Liera : mcastroliera@gmail.com

        This program is free software; you can redistribute it and/or modif
        it under the terms of the GNU General Public License as published b
        the Free Software Foundation; either version 2 of the License, or
        (at your option) any later version.
        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
        GNU General Public License for more details.
        You should have received a copy of the GNU General Public License
        along with this program; if not, write to the Free Software
        Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
        MA 02110-1301, USA.
*/

//Cabeceras y librerias necesarias

#include <stdio.h>
#include <math.h>
#include "../instancias/tsp225.h"
#include "params.h"
#include "alea.h"
#include "mpi.h"
#include "busqlocal.h"

//Estructura para los individuos.
typedef struct
```

```
{
  int p[NC];
  int aptitud;
} individuo;

//Variables globales necesarias.
individuo poblacion[TAM_POB], best, actual;
int siguiente;

//Metodo que construye la matriz de distancias dist a partir de la matriz t
void calculadistancias(void)
{
  int i,j;
  double dx,dy;

  for(i=0; i<NC; i++)
  {
    for(j=0; j<i; j++)
    {
      dx=tsp225[i][0]-tsp225[j][0];
      dy=tsp225[i][1]-tsp225[j][1];
      dist[i][j]=dist[j][i]=round(sqrt(dx*dx+dy*dy));
    }
    dist[i][i]=0;
  }
}

//Metodo que calcula la aptitud de una permutacion.
int aptitud(int *p)
{
  int i;
  int suma=0;
  for(i=1; i<NC; i++)
  {
    suma+=dist[p[i-1]][p[i]];
  }
  suma+=dist[p[NC-1]][p[0]];
  return suma;
}

//Metodo que lleva el elemento indice de la poblacion a un optimo local.
void busquedalocal(int indice)
{
        //Primero aplica un algoritmo 2-opt y posteriormente un 3-opt optim
        dosopt(poblacion[indice].p);
        TresOptP(poblacion[indice].p);
}
```

```
//Metodo que inicializa la poblacion.
void inicializa_poblacion(void)
{
  int i,j;
  for(i=0; i<TAM_POB; i++)
  {
    for(j=0; j<NC; j++)
    {
      poblacion[i].p[j]=j;
    }
    //Se barajea la permutacion y posteriormente se le aplica el algoritmo
    gsl_ran_shuffle(r,poblacion[i].p,NC,sizeof(int));
    busquedalocal(i);
    //Se calcula su aptitud y se verifica que no sea la mejor.
    poblacion[i].aptitud=aptitud(poblacion[i].p);
    if(i==0||poblacion[i].aptitud<best.aptitud)
    {
      best=poblacion[i];
    }
  }
}
//Metodo que ordena la poblacion mediante el algoritmo Quicksort
void ordenar(int lim_iz,int lim_de)
{
  int izq, der;
  individuo temp, piv;

  izq=lim_iz;
  der=lim_de;
  piv=poblacion[(izq+der)/2];

  do
  {
    while(poblacion[izq].aptitud<piv.aptitud&&izq<lim_de)
    {
      izq++;
    }
    while(piv.aptitud < poblacion[der].aptitud&&der>lim_iz)
    {
      der--;
    }
    if(izq<=der)
    {
      temp=poblacion[izq];
      poblacion[izq]=poblacion[der];
      poblacion[der]=temp;
```

```
      izq++;
      der--;
    }
  }
  while(izq<=der);

  if(lim_iz<der)
  {
    ordenar(lim_iz, der);
  }
  if(lim_de>izq)
  {
    ordenar(izq,lim_de);
  }
}
//Metodo que implementa la cruza OX para permutaciones.
void cruce(void)
{
  int i,j,k,q,pc,p1,p2,bandera;
  int inicio,fin,aux;
  //Se genera un punto de corte, a partir del cual dividimos la poblacion e
  pc=TAM_POB*PS;
  for(i=pc; i<TAM_POB-1; i++)
  {
    p1=ZNM(pc);
    do
    {
      p2=ZNM(pc);
    }
    //p1 y p2 son los indices dentro de la poblacion de los padres.
    while(p1==p2);
    inicio=ZNM(NC);
    do
    {
      fin=ZNM(NC);
    }
    while(inicio==fin);
    if(inicio>fin)
    {
      aux=inicio;
      inicio=fin;
      fin=aux;
    }
    //Seccion que construye el primer hijo.
    //Primero se copian los elementos de las posiciones comprendidas entre
    for(j=inicio; j<=fin; j++)
    {
```

```
  poblacion[i].p[j]=poblacion[p1].p[j];
}
q=0;
//Posteriormente se copian los elementos del segundo padre que no se en
for(j=0; j<inicio; j++)
{
  do
  {
    bandera=0;
    for(k=inicio; k<=fin; k++)
    {
      if(poblacion[p2].p[q]==poblacion[p1].p[k])
      {
        bandera=1;
        q++;
        k=NC;
      }
    }
  }
  while(bandera==1);
  poblacion[i].p[j]=poblacion[p2].p[q];
  q++;
}
for(j=fin+1; j<NC; j++)
{
  do
  {
    bandera=0;
    for(k=inicio; k<=fin; k++)
    {
      if(poblacion[p2].p[q]==poblacion[p1].p[k])
      {
        bandera=1;
        q++;
        k=NC;
      }
    }
  }
  while(bandera==1);
  poblacion[i].p[j]=poblacion[p2].p[q];
  q++;
}
//Sobre el nuevo individuo de aplica busqueda local.
busquedalocal(i);
poblacion[i].aptitud=aptitud(poblacion[i].p);
if(poblacion[i].aptitud<best.aptitud)
{
```

```
    best=poblacion[i];
  }
  //Seccion que construye el segundo hijo, es analoga a la del primer hij
  i++;
  for(j=inicio; j<=fin; j++)
  {
    poblacion[i].p[j]=poblacion[p2].p[j];
  }
  q=0;
  for(j=0; j<inicio; j++)
  {
    do
    {
      bandera=0;
      for(k=inicio; k<=fin; k++)
      {
        if(poblacion[p1].p[q]==poblacion[p2].p[k])
        {
          bandera=1;
          q++;
          k=NC;
        }
      }
    }
    while(bandera==1);
    poblacion[i].p[j]=poblacion[p1].p[q];
    q++;
  }
  for(j=fin+1; j<NC; j++)
  {
    do
    {
      bandera=0;
      for(k=inicio; k<=fin; k++)
      {
        if(poblacion[p1].p[q]==poblacion[p2].p[k])
        {
          bandera=1;
          q++;
          k=NC;
        }
      }
    }
    while(bandera==1);
    poblacion[i].p[j]=poblacion[p1].p[q];
    q++;
  }
```

```
    busquedalocal(i);
    poblacion[i].aptitud=aptitud(poblacion[i].p);
    if(poblacion[i].aptitud<best.aptitud)
    {
      best=poblacion[i];
    }
  }
}

//Implementa una mutacion con un intercambio de cuatro aristas sobre la peri
void mutacion(void)
{
        int iti,itj, ban;
        int aux;
        int ind,i,j,k,l;
        //Ciclo para elegir que individuo mutar, se excluyen los ELT mejore
        for(ind=ELT; ind<TAM_POB; ind++)
        {
                if(UNI<PM)
                {
                        //Se generan cuatro puntos de corte, y se invierten

                        i = OTN(floor(NC/2));
                        j = i + 1 + OTN(floor(NC/4));
                        k = j + 1 + OTN(floor(NC/8));
                        l = k + 1 + OTN(NC-k-2);
                        //Ciclo para invertir segmento i-j
                        for(iti=i, itj=j; iti<itj; iti++, itj--)
                        {
                                aux = poblacion[ind].p[iti];
                                poblacion[ind].p[iti] = poblacion[ind].p[it
                                poblacion[ind].p[itj] = aux;
                        }
                        //Ciclo para invertir segmento j-k
                        for(iti=j, itj=k; iti<itj; iti++, itj--)
                        {
                                aux = poblacion[ind].p[iti];
                                poblacion[ind].p[iti] = poblacion[ind].p[it
                                poblacion[ind].p[itj] = aux;
                        }
                        //Ciclo para invertir segmento k-l
                        for(iti=k, itj=l; iti<itj; iti++, itj--)
                        {
                                aux = poblacion[ind].p[iti];
                                poblacion[ind].p[iti] = poblacion[ind].p[it
                                poblacion[ind].p[itj] = aux;
                        }
```

```
                        //Ciclo para invertir segmento l-i
                        ban=0;
                        for(iti=l, itj=i; (iti<itj || ban==0); iti++, itj--
                        {
                                if(iti==NC)
                                {
                                        iti=0;
                                        ban=1;
                                }
                                if(itj<0)
                                {
                                        itj=NC-1;
                                        ban=1;
                                }
                                aux = poblacion[ind].p[iti];
                                poblacion[ind].p[iti] = poblacion[ind].p[it
                                poblacion[ind].p[itj] = aux;
                        }
                        //Una vez que se llevo a cabo el double-bridge, se
                        busquedalocal(ind);
                        poblacion[ind].aptitud=aptitud(poblacion[ind].p);
                        if(poblacion[ind].aptitud<best.aptitud)
                        {
                                best=poblacion[ind];
                        }
                }
        }
}
//Metodo que envia los mejores TM individuos a la poblacion vecina.
void envia(void)
{
  int i;
  for(i=0; i<TM; i++)
  {
    MPI_Send(&poblacion[i].aptitud,1,MPI_INT,siguiente,21,MPI_COMM_WORLD);
    MPI_Send(poblacion[i].p,NC,MPI_INT,siguiente,12,MPI_COMM_WORLD);
  }
}
//Metodo que recibe los mejores TM individuos a la poblacion vecina.
void recibe(void)
{
  int i;
  for(i=1; i<=TM; i++)
  {
    MPI_Recv(&poblacion[TAM_POB-i].aptitud,1,MPI_INT,MPI_ANY_SOURCE,21,MPI_
    MPI_Recv(poblacion[TAM_POB-i].p,NC,MPI_INT,MPI_ANY_SOURCE,12,MPI_COMM_W
  }
```

```
  //En caso de recibir un individuo mejor que el mejor de la poblacion, se
  if(poblacion[TAM_POB-1].aptitud<best.aptitud)
  {
    best=poblacion[TAM_POB-1];
  }
}

int main(int argc, char *argv[])
{
  //Parametros generales del programa.
  int g,muestra;
  int nproc, iproc; //nproc numero de procesos, iproc mi indice de proceso
  double ti,tf;
  //Parametros de MPI.
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  MPI_Comm_rank(MPI_COMM_WORLD,&iproc);
  //El proceso con iproc=0 es el maestro.
  if(iproc>0)
  {
    //Cada proceso exclavo primero determina a que proceso va a migrar.
    siguiente = (iproc<nproc-1) ? iproc+1 : 1;
    calculadistancias();
    //Cada muestra es una ejecucion del algoritmo.
    for(muestra=0; muestra<TMUESTRA; muestra++)
    {
      //Primero inicializamos el generador de aleatorios y la poblacion.
      inirndm();
      inicializa_poblacion();
      //Ciclo principal del algoritmo memetico.
      for(g=0; g<GMAX; g++)
      {
        ordenar(0,TAM_POB-1);
        //Si la generacion es multiplo de MP, migramos con los procesos vec
        if(g!=0&&g%MP==0)
        {
          envia();
          recibe();
          ordenar(0,TAM_POB-1);
        }
        //Operadores geneticos.
        cruce();
        mutacion();
      }
      //Al terminar el ciclo principal, se envia el mejor individuo al proc
      MPI_Send(&best.aptitud,1,MPI_INT,0,73,MPI_COMM_WORLD);
      MPI_Send(best.p,NC,MPI_INT,0,37,MPI_COMM_WORLD);
```

```
    }
  }
  else
  {
    for(muestra=0; muestra<TMUESTRA; muestra++)
    {
      ti=MPI_Wtime();
      //Ciclo en el cual el proceso maestro recibe los mejores individuos d
      for(g=1; g<nproc; g++)
      {
        MPI_Recv(&actual.aptitud,1,MPI_INT,MPI_ANY_SOURCE,73,MPI_COMM_WORLD
        MPI_Recv(actual.p,NC,MPI_INT,MPI_ANY_SOURCE,37,MPI_COMM_WORLD,MPI_S
        if(g==1||best.aptitud>actual.aptitud)
        {
          best=actual;
        }
      }
      //Se despliega en pantalla el mejor individuo de todos los procesos e
      tf=MPI_Wtime();
      printf("%04d␣␣%04d␣%f\n",muestra+1,best.aptitud,tf-ti);
    }
  }
  MPI_Finalize();
  return 0;
}
```

8.2. Cabeceras

8.2.1. Cabecera con la implementación de la búsqueda local

```
/*
        busqlocal.h, archivo de encabezado donde se encuentran las funcione

        Copyrigth (c) 2016 Joel Artemio Morales Viscaya : iscviscaya@gmail.
        Marco Antonio Castro Liera : mcastroliera@gmail.com

        This program is free software; you can redistribute it and/or modif
        it under the terms of the GNU General Public License as published b
        the Free Software Foundation; either version 2 of the License, or
        (at your option) any later version.
        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
        GNU General Public License for more details.
```

```
        You should have received a copy of the GNU General Public License
        along with this program; if not, write to the Free Software
        Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
        MA 02110-1301, USA.
*/

//Estructura arista, necesaria para la estrategia 3-Opt implementada.
typedef struct
{
  int n1;
  int n2;
  int costo;
} arista;

//Metodo que implementa 2-opt sobre la permutacion
void dosopt(int *t)
{
  int i,j,mi,mj,ganancia,actual;
  do
  {
    mi=mj=ganancia=0;
    for(i=1; i<NC-1; i++)
    {
      for(j=i+1; j<NC; j++)
      {
        actual = (dist[t[i-1]][t[i]]+dist[t[j]][t[(j+1)%NC]]) -
                 (dist[t[i-1]][t[j]]+dist[t[i]][t[(j+1)%NC]]);
        if(actual>ganancia)
        {
          ganancia=actual;
          mi=i;
          mj=j;
        }
      }
    }
    while(mi<mj)
    {
      actual=t[mi];
      t[mi]=t[mj];
      t[mj]=actual;
      mi++;
      mj--;
    }
  }
  while(ganancia>0);
}
```

```
//Metodo que verifica que una arista en particular no se encuentra en un ar
int NoContiene(arista* lista, int p, int q)
{
    int i, bandera;
    bandera=0;
    for(i=0; i<NC; i++)
    {
        if((lista[i].n1==p && lista[i].n2==q) || (lista[i].n2==p && lista[i
        {
            bandera=1;
        }
    }
    return bandera;
}

//Metodo que busca en que posicion de la arista se encuentra un valor val,
int Busca(arista* T, int val, int tabu)
{
    int i, retorno=0;
    for(i=0; i<NC; i++)
    {
        if(i!=tabu && (T[i].n1==val || T[i].n2==val))
        {
            retorno=i;
            break;
        }
    }
    return retorno;
}

//Metodo que verifica que un conjunto de aristas K, sea Hamiltoniano
int EsHamiltoniano(arista* K)
{
    int Lista[NC];
    int i,bandera, posactual;
    arista T[NC];
    for(i=0; i<NC; i++)
    {
        T[i].n1=K[i].n1;
        T[i].n2=K[i].n2;
    }
    //Hay que revisar si T es un recorrido hamiltoniano.
    for(i=0; i<NC; i++)
    {
        Lista[i]=0;
    }
    Lista[T[0].n1]=1;
```

```
    posactual=0;
    bandera=1;
    for(i=0; i<NC-1; i++)
    {
        if(Lista[T[posactual].n1]==1)
        {
            if(Lista[T[posactual].n2]==1)
            {
                bandera=0;
                break;
            }
            else
            {
                //posactual igual a donde este T[posactual].n2
                Lista[T[posactual].n2]=1;
                posactual=Busca(T,T[posactual].n2, posactual);
            }
        }
        else
        {
            Lista[T[posactual].n1]=1;
            posactual=Busca(T,T[posactual].n1, posactual);
        }
    }
    return bandera;
}


//Metodo que despliega el conjunto de aristas como un recorrido de ciudades
void RegresaRuta(arista* T,int *Ruta)
{
    int posactual,i;
    int Lista[NC];
    for(i=0; i<NC; i++)
    {
        Lista[i]=0;
    }
    Lista[T[0].n1]=1;
    posactual=0;
    //printf("{%d-",T[0].n1);
    Ruta[0]=T[0].n1;
    for(i=0; i<NC-1; i++)
    {
        if(Lista[T[posactual].n1]==1)
        {
            Lista[T[posactual].n2]=1;
            Ruta[i+1]=T[posactual].n2;
```

```
            posactual=Busca(T,T[posactual].n2, posactual);
        }
        else
        {
            Lista[T[posactual].n1]=1;
            Ruta[i+1]=T[posactual].n1;
            posactual=Busca(T,T[posactual].n1, posactual);
        }
    }
}

//Metodo que implementa el algoritmo de busqueda local propuesto.
void TresOptP(int* Ruta)
{
//Declaracion de variables.
    int i,j,jj,pos,k,pos1,pos2,contador=0;
    int dos=0;
    int posY=-1;
    arista x[3];
    arista y[3];
    arista Q[NC];
    double ganancia=0;
    int Lista3[NC];
    int ListaX[NC];
    int ListaY[NC];
    int posX=NC;
    int posP=NC;
    arista T[NC];
    //Inicializacion de aristas, primero asignamos un costo a las aristas x
    for(i=0; i<3; i++)
    {
        x[i].n1=x[i].n2=x[i].costo=0;
    }
    //Convertir Ruta en arreglo de aristas llamada T, guardamos una copia e
    for(i=0; i<NC; i++)
    {
        ListaX[i]=0;
        Lista3[i]=0;
        ListaY[i]=0;
        Q[i].n1 = T[i].n1 = Ruta[i];
        Q[i].n2 = T[i].n2 = (i==NC-1) ? Ruta[0]:Ruta[i+1];
        Q[i].costo = T[i].costo = dist[T[i].n1][T[i].n2];
    }
//Inicio del bloque principal
ini:
    //Checamos si es optimo local.
    if(posP==0)
```

```
    {
        goto fin;
    }
    //Si no estamos en un optimo local, eleguimos aleatoriamente una arista
    i = ZNM(posP);
    for(j=0; j<=i; j++)
    {
        if(ListaX[j]!=0)
        {
            i++;
        }
    }
    x[0]=T[i];
    //Una vez generado elegido x[0], se toma y[0], arista a agregar, tal qu
    //que x[0].
pin:
    posY=-1;    //posY se utiliza para determinar en que ocasiones no se en
    for(j=0; j<NC; j++)
    {
        if(dist[x[0].n1][j]<x[0].costo && NoContiene(T,x[0].n1,j)==0 && dis
        {
            //Elegimos y[0]
            y[0].n1 = x[0].n1;
            y[0].n2 = j;
            posY = j;
            y[0].costo = dist[y[0].n1][y[0].n2];
            //y[0] la arista que une x[0].n1 con la ciudad j, una vez se en
            break;
        }
    }
    if(posY==-1)
    {
        //No hay ninguna y[0] disponible para la x[0] eleguida aleatoriamen
        //Se resetean todas las listas y se tacha el x[0].
        posX=NC;
        for(j=0; j<NC; j++)
        {
            Lista3[j]=0;
            ListaY[j]=0;
        }
        posP--;
        ListaX[i]=1;
        goto ini;       //Regresamos al comienzo del algoritmo, con la x[0]
    }
    //Si no hubo inconvenientes y se encontro y[0] satisfactoria, continuam
    //Buscamos donde aparece y[0].n2 en T.
    pos1 = Busca(T,y[0].n2,NC);
```

```
    pos2 = Busca(T,y[0].n2,pos1);
    posX = posX-2;
bandos:
    //Elegimos una arista que contenga a y[0].n2 para ser nuestra x[1], con
    if((T[pos1].costo/(T[pos1].costo+T[pos2].costo))>UNI)
    {
        if(T[pos1].n1==y[0].n2)
        {
            T[pos1].n1=T[pos1].n2;
            T[pos1].n2=y[0].n2;
        }
        pos=pos1;
    }
    else
    {
        if(T[pos2].n1==y[0].n2)
        {
            T[pos2].n1=T[pos2].n2;
            T[pos2].n2=y[0].n2;
        }
        pos=pos2;
    }
    //En pos quedo la posicion adecuada.
    x[1]=T[pos];
    Lista3[i]=1;
    Lista3[pos]=1;
ban:
    k = ZNM(posX);      //Se eligue x[2] de manera aleatoria entre las aris
    for(j=0; j<=k; j++)
    {
            if(Lista3[j]!=0)
            {
                k++;
            }
    }
    x[2] = T[k];
    //Ya con x[0],x[1],x[2],y[0], las posibilidades de y[1] y y[2] son limi
    //Probamos cada una, la que produzca un tour con menor costo que el ori
    //sea recorrido hamiltoniano se elige.
    for(jj=0; jj<2; jj++)
    {
            if(jj==0)   //Una eleccion posible en la primera vuelta.
            {
                    y[1].n1=x[2].n1;
                    y[1].n2=x[1].n1;
                    y[2].n1=x[2].n2;
                    y[2].n2=x[0].n2;
```

```
        }
        else
        {           //La otra eleccion posible en la segunda vuelta.
                y[1].n1=x[2].n1;
                y[1].n2=x[0].n2;
                y[2].n1=x[2].n2;
                y[2].n2=x[1].n1;
        }
        y[1].costo = dist[y[1].n1][y[1].n2];
        y[2].costo = dist[y[2].n1][y[2].n2];
        ganancia=(x[0].costo+x[1].costo+x[2].costo)-(y[0].costo+y[1].co
            //Calculamos la ganancia de cambiar las aristas x por las a
        if(ganancia>0)
        {
            Q[i]=y[0];
            Q[pos]=y[1];
            Q[k]=y[2];
                    //Si la ganancia es positiva, verificamos que nos p
            if(EsHamiltoniano(Q)==1)
            {
                T[i]=y[0];
                T[pos]=y[1];
                T[k]=y[2];
                for(j=0; j<NC; j++)
                {
                    ListaX[j]=0;
                    Lista3[j]=0;
                    ListaY[j]=0;
                }
                posX=NC;
                posP=NC;
                contador++;
                goto ini;
            }
            else
            {
                Q[i]=T[i];
                Q[pos]=T[pos];
                Q[k]=T[k];
            }
        }
}
//Si se probo con las dos opciones y ninguna encontro un tour mejor que
//entonces tachamos el x[2]
Lista3[k]=1;
posX--;
//Si posX no es cero todavia, se selecciona otro x[2].
```

```
    if(posX!=0)
    {
        goto ban;       //Brincamos a la eleccion aleatoria de x[2], probam
    }
    else
    {
        //Si posX ya fue cero, entonces no hay un x[2] adecuado, buscamos o
        //Reseteamos toda la lista tachada para x[2]
        posX=NC;
        for(j=0; j<NC; j++)
        {
            Lista3[j]=0;
        }
        //Tachamos ahora la y[0]
        ListaY[posY] = 1;
        posY=-1;
        goto pin;       //Brincamos a la parte posterior a la eleccion de x
    }
fin:
    RegresaRuta(T,Ruta);        //Salida del programa.
}
```

8.2.2. Cabecera con los parámetros del programa, params.h

```
//params.h

#define TMUESTRA 100    //Tamano de la muestra
#define GMAX     20             //Numero de generaciones
#define TAM_POB 13              //Tamano de la poblacion
#define PS       0.4            //Presion de seleccion
#define PM       0.1            //Probabilidad de mutacion
#define ELT      1              //Elitismo
#define MP       3                      //Periodo migratorio
#define TM       1                      //Razon de migracion

int dist[NC][NC];               //Matriz con las distancias de las ciudades
```

8.2.3. Cabecera para la generación de aleatorios, alea.h

```
//alea.h

#include <gsl/gsl_rng.h>
#include <sys/time.h>
```

```
#include <gsl/gsl_randist.h>

#define UNI gsl_rng_uniform_pos(r)      //Aleatorio entre (0,1)
#define VNI (1-2*UNI)   //Aleatorio entre (-1,1)
#define NNI(n) (n*VNI)  //Aleatorio flotante entre (0,n)
#define ZNM(n) gsl_rng_uniform_int(r,n)         //Aleatorio entero entre 0
#define OTN(n) (ZNM(n)+1)       //Aleatorio entero entre 1 y n.

gsl_rng * r;

//Funcion que inicializa el generador de aleatorios.
void inirndm(void)
{
  struct timeval tv;
  struct timezone tz;
  gettimeofday(&tv,&tz);
  r = gsl_rng_alloc(gsl_rng_mt19937);
  gsl_rng_set(r,tv.tv_usec);
}
```