UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

ALGORITMOS MODIFICADOS PARA RESOLVER EL PROBLEMA DEL AGENTE VIAJERO: BÚSQUEDA LOCAL, TEMPLADO SIMULADO Y MÁQUINA DE BOLTZMANN SECUENCIAL.

T E S I S

QUE PARA OBTENER EL TÍTULO DE

M A T E M Á T I C O

P R E S E N T A :

LUIS CARMONA VILLANUEVA

DIRECTOR DE TESIS

DR. PABLO BARRERA SÁNCHEZ

FACULTAD DE CIENCIAS
UNAM

2005

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales de la Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

Algoritmos modificados para resolver el problema del agente viajero: búsqueda local, templado simulado, máquina de boltzmann secuencial.

realizado por Carmona Villanueva Luis

con número de cuenta 09455886-7 , quien cubrió los créditos de la carrera de: Matemáticas

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	Dr. Pablo Barrera Sánchez
Propietario	Dr. Jesús López Estrada
Propietario	M. en C. María de Luz Gasca Soto
Suplente	M. en C. María Lourdes Velasco Arregui
Suplente	M. en C. María Guadalupe Elena Ibarguengoytia González

Consejo Departamental de
Matemáticas

M. en C. Alejandro Bravo Mojica

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMATICAS

Índice General

Introducción

En este trabajo revisaremos tres técnicas de solución para resolver problemas de optimización combinatoria. Específicamente estudiaremos e implementaremos los algoritmos búsqueda local, templado simulado y máquina de Boltzmann secuencial para resolver el problema del agente viajero.

Nuestro objetivo es realizar modificaciones a los algoritmos antes mecionados para mejorar la calidad de las soluciones encontradas.

Para probar los algoritmos (originales y modificados) resolveremos el problema viaje redondo por 51 ciudades de México. El problema anterior es un ejemplo específico del problema del agente viajero.

En el Capítulo 1 revisaremos primero las definiciones que proporcionan la teoría de gráficas y la programación lineal para el problema del agente viajero. Después estudiaremos algunos conceptos de optimización combinatoria. Dichos conceptos nos permitirán redefinir al problema del agente viajero para trabajarlo como un problema de optimización combinatoria.

En el Capítulo 2 detallaremos los algoritmos búsqueda local, templado simulado y máquina de Boltzmann secuencial. Para cada algoritmo primero presentamos la técnica general y después la técnica enfocada a resolver el problema del agente viajero.

En el Capítulo 3 mostraremos nuestras propuestas de modificación para los algoritmos búsqueda local, templado simulado y máquina de Boltzmann secuencial.

En el Capítulo 4 resolveremos el problema viaje redondo por 51 ciudades de México. Para la solución de dicho problema utilizaremos los tres algoritmos mencionados y sus respectivas modificaciones. La codificación de los algoritmos la realizaremos en lenguaje C [1]. Las pruebas las realizaremos en una PC con procesador a 600 Mz y en ambiente Linux. Para realizar la comparación de los algoritmos calcularemos su error relativo y su orden de complejidad. Finalizaremos este capítulo con la mejor solución encontrada por los algoritmos para el problema viaje redondo por 51 ciudades de México.

[1] Los códigos fuente aparecen en el Apéndice C.

Capítulo 1

Problema del Agente Viajero

De manera general el *problema del agente viajero* es descrito como sigue. Dada una lista de ciudades y las distancias entre cada par de ellas, un agente viajero desea realizar un viaje redondo visitando exactamente una vez cada ciudad, y de tal forma que la distancia total de su viaje sea la más pequeña posible [11].

Algunas áreas de las matemáticas que han abordado el problema del agente viajero son:

- la teoría de gráficas,
- la programación lineal y
- la optimización combinatoria.

En las siguientes secciones analizaremos las definiciones para el problema del agente viajero que nos dan cada una de las áreas antes mencionadas.

1.1 Teoría de Gráficas.

En esta sección nos enfocaremos en la definición del problema del agente viajero que nos proporciona la teoría de gráficas. Para mostrar tal definición es necesario revisar algunas definiones básicas de la teoría de gráficas.

1.1.1 Definiciones

Una *gráfica* $G = (V, A)$ es un conjunto no vacío de puntos V y un conjunto A de pares de esos puntos. Los elementos de V son llamados *vértices* y los elementos de A son llamados *arcos* [7].

Un arco es representado como el par (i, j) donde i es el vertice inicial y j es el vertice final [7]. Para simplificar la notación utilizaremos simplemente ij.

Algunas veces un número c_{ij} es asociado con el arco ij. Dicho número es llamado el *peso* del arco [7]. Otros nombres son la *longitud* o el *costo* del arco.

Una gráfica con pesos en sus arcos es llamada *gráfica ponderada* [7].

Dada una gráfica $G = (V, A)$, un *camino* entre los vértices u y z es una sucesión de arcos de la forma $uv, vw, wx, ..., yz$. Pueden existir varios caminos para cada par de vértices.[3]

Un *camino cerrado,* en una gráfica, es una sucesión de arcos que comienza y termina en el mismo vértice, es decir de la forma:

$$ij, jp, pq, ..., uv, vi$$

Un *ciclo*, en una gráfica, es un camino cerrado en el cual todos los arcos y todos los vértices intermedios son diferentes [3].

En una grafica ponderada, la *longitud de un ciclo* es la suma de los pesos de cada uno de sus arcos.

Un *ciclo hamiltoniano*, en una gráfica, es un ciclo que contiene todos los vértices de la gráfica [3].

1.1.2 El Problema del Agente Viajero

Sea n el número de ciudades, las distancias entre cada par de ciudades se pueden organizar en una matriz $n \times n$, es decir:

$$D = \begin{vmatrix} d_{11} & d_{12} & \cdots & d_{1n} \\ d_{21} & d_{22} & \cdots & d_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n1} & d_{n2} & \cdots & d_{nn} \end{vmatrix}$$

Donde d_{ij} es la distancia de la ciudad i a la ciudad j.

Las n ciudades pueden ser representadas como vértices y los caminos entre cada par de ciudades como arcos. Entonces el problema del agente viajero puede ser representado como una gráfica ponderada $G = (V, A)$ donde el peso del arco ij es d_{ij}. Considerando lo anterior tenemos la siguiente definición del problema del agente viajero como un problema en gráficas.

Dada una gráfica ponderada $G = (V, A)$, el *problema del agente viajero* consiste en encontrar un ciclo hamiltoniano H de longuitud mínima [3].

1.2 Programación Lineal

Una segunda definición del problema del agente viajero nos la proporciona la programación lineal. Para mostrar tal definición necesitamos revisar algunos conceptos.

1.2.1 Definiciones

Un *problema de programación lineal* consiste en encontrar un vector en R^n que optimice (maximice o minimice) una función f (de R^n a R); además el vector debe de satisfacer un conjunto finito de desigualdades lineales. Tal vector es llamado *solución optima global*, el conjunto de soluciones que satisfacen el conjunto de desigualdades es llamado *conjunto de soluciones factibles* y f es denominada *función de costo* [17] [20].

Un problema de programación lineal es llamado *problema de programación lineal 01* si sus soluciones factibles (vectores en R^n) son definidas como vectores binarios.

1.2.2 El Problema del Agente Viajero

Sea n es el número de ciudades, las distancias entre cada par de ciudades se pueden organizar en una matriz $n \times n$, es decir:

$$D = \begin{vmatrix} d_{00} & d_{01} & \cdots & d_{0(n-1)} \\ d_{10} & d_{11} & \cdots & d_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n0} & d_{n1} & \cdots & d_{(n-1)(n-1)} \end{vmatrix}$$

Donde d_{ij} es la distancia de la ciudad i a la ciudad j.

Una solución puede ser dada como una matriz binaria $n \times n$, $X = [x_{ip}]$, definida por :

$$x_{ip} = \begin{cases} 1 & \text{si la ciudad } i \text{ es visitada en la posición } p \\ 0 & \text{en otro caso} \end{cases}$$

Con base en lo anterior, tenemos la definición del problema de agente viajero como un problema de programación lineal 01.

Dada una matriz $n \times n$ de distancias, $D = [d_{ij}]$, el *problema del agente viajero* consiste en encontrar una matriz binaria $n \times n$, $X = [x_{ij}]$, tal que:
minimice

$$f(X) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{p=0}^{n-1} \sum_{q=0}^{n-1} a_{ijpq} \cdot x_{ip} \cdot x_{jq},$$

sujeto a

$$\sum_{i=0}^{n-1} x_{ip} = 1, \qquad p = 0, ..., n-1;$$
$$\sum_{p=0}^{n-1} x_{ip} = 1, \qquad i = 0, ..., n-1.$$

Donde a_{ijpq} es un número real calculado como:

$$a_{ijpq} = \begin{cases} d_{ij} & \text{si } q = (p+1) \bmod n; \\ 0 & \text{en otro caso} \end{cases}$$

El valor de la función $f(X)$ es igual a la distancia total del viaje X [1].

1.3 Problemas de Optimización Combinatoria

La investigación en el área de *optimización combinatoria* (o discreta) está dirigida a desarrollar técnicas eficientes que permitan encontrar valores mínimos o máximos de una función (función de costo) multivariable [15].

Las siguientes definiciones son necesarias para mostrar como las dos definiciones anteriores del problema del agente viajero (teoría de gráficas y programación lineal) se pueden formular como problemas de optimización combinatoria.

1.3.1 Definiciones

Un problema de *optimización combinatoria* es definido como el conjunto de sus ejemplares [17].

El término ejemplar es utilizado para distinguir una situación particular de una situación general [18]. Por ejemplo, si tomamos como situación general el concepto "libro", entonces "esta tesis" es un ejemplar de un "libro".

Un *ejemplar de un problema de optimización combinatoria* es una terna (E, S, f), donde:

- E y S son conjuntos de vectores de variables discretas (por ejemplo enteros).
- S es un subconjunto finito de E, $S \subseteq E$
- f es una función de S a los reales, $f : S \longrightarrow R$

Al conjunto E se le llama *espacio de soluciones*, al conjunto S se le llama conjunto de *soluciones factibles* y a la función f se le llama *función de costo* [17] [18] [20] .

Sea (E, S, f) un ejemplar de un problema de optimización combinatoria, $x \in S$ y $c = f(x)$. Entonces c es llamado el *costo* de la solución x.

Sea (E, S, f) un ejemplar de un problema de optimización combinatoria, entonces:

- una *solución mínima global* es una solución $x_{min} \in S$ tal que

$$f(x_{min}) \leq f(y) \text{ para toda } y \in S;$$

- una *solución máxima global* es una solución $x_{max} \in S$ tal que

$$f(x_{max}) \geq f(y) \text{ para toda } y \in S;$$

- una solución minima global o una solución máxima global es llamada, de manera general, *solución óptima global* y es denotada por x_{opt}.

Así, resolver un problema de optimización combinatoria consiste en encontrar una solución óptima global [17] [18] [20].

Sea (E, S, f) un ejemplar de un problema de optimización combinatoria y $x \in S$. Si a la solución x se le borra, agrega o intercambia un elemento se dice que se le ha aplicado una *operación simple* [20]. Denotaremos como σ a cualquiera de estas operaciones.

Si las soluciones están dadas por vectores binarios, entonces una operación simple puede ser por ejemplo cambiar a 0 una entrada que valga 1.

Sea (E, S, f) un ejemplar de un problema de optimización combinatoria. Una *estructura de vecindad* $\mathcal{N}(x, \sigma)$ es una función

$$\mathcal{N} : S \longrightarrow 2^S$$

la cual define para cada solución $x \in S$ un conjunto S_x de soluciones que pueden ser alcanzadas a partir de x por medio de la operación simple σ. El conjunto S_x es llamado una *vecindad* de la solución x [20] [17].

Sea (E, S, f) un ejemplar de un problema de optimización combinatoria y $\mathcal{N}(x, \sigma)$ una estructura de vecindad, entonces:

- una *solución mínima local* es una solución $\widehat{x} \in S$ tal que

$$f(\widehat{x}) \leq f(y) \text{ para toda } y \in S_x;$$

- una *solución máxima local* es una solución $\widehat{x} \in S$ tal que

$$f(\widehat{x}) \geq f(y) \text{ para toda } y \in S_x;$$

- una solución mínima local o una solución máxima local es llamada, de manera general, *solución óptima local* [17].

1.3.2 El Problema del Agente Viajero

La definición del problema del agente viajero de la teoría de gráficas se puede formular como un problema de optimización combinatoria. La siguiente definición nos muestra el hecho anterior.

Dada una gráfica ponderada $G = (V, A)$, el *problema del agente viajero* consiste encontrar una solución mínima global de (E, S, f), donde:

- $E = \{\text{ciclos de } G\}$;
- $S = \{x \in E \mid x \text{ es un ciclo hamiltoniano}\}$;
- $f(x) = \sum c_{ij}$ tal que el arco $ij \in x$.

A continuación mostramos como la definición del problema del agente viajero visto en la programación lineal se puede formular como un problema de optimización combinatoria.

Dada una matriz $n \times n$ de distancias, $D = [d_{ij}]$, el *problema del agente viajero* consiste en encontrar una solución mínima global de (E, S, f), donde:

- $E = \{\text{matrices binarias de } n \times n\}$;
- $S = \left\{ x \in E \mid \sum_i x_{ip} = 1 \text{ y } \sum_p x_{ip} = 1 \right\}$;
- $f(x) = \sum_i \sum_j \sum_p \sum_q a_{ijpq} \cdot x_{ip} \cdot x_{jq}$.

Capítulo 2

Algoritmos para el Problema del Agente Viajero

De manera general un *problema* es una situación en la que las cosas que tenemos son diferentes de las que deseamos. Por ejemplo si queremos un pastel y tenemos los ingredientes para hacerlo, entonces tenemos un problema.

Un *algoritmo* es una serie de pasos ordenados lógicamente que permiten resolver un problema. En otras palabras un algoritmo transforma lo que tenemos en lo que deseamos. En el ejemplo anterior una receta para hacer un pastel es un algoritmo.

En este capítulo trataremos algoritmos diseñados para resolver el problema del agente viajero. Dado que queremos resolver nuestro problema utilizando una computadora nos enfocaremos en algoritmos pertenecientes al área de ciencias de la computación.

2.1 Problemas y Algoritmos Computacionales

Para clasificar el problema del agente viajero dentro del área de ciencias de la computación revisaremos primero algunas definiciones generales. Dicha casificación nos permitirá comprender la naturaleza de los algoritmos diseñados para resolver nuestro problema.

2.1.1 Definiciones

Los problemas que pueden ser resueltos utilizando una computadora son llamados *problemas computacionales*. Para estos problemas "las cosas que tenemos" son llamadas *datos de entrada* y "las cosas que deseamos" son llamadas *datos de salida*. Llamaremos *ejemplar de un problema* a una asignación particular de los datos de entrada [6].

Los problemas computacionales se clasifican en *problemas de requerimientos* y *problemas de dificultad* [19].

Los problemas de requerimientos se dividen en:

- *Problemas de búsqueda*: Encontrar un valor, en los datos de entrada, que satisfaga una propiedad.
- *Problemas de estructura*: Transformar los datos de entrada para satisfacer una propiedad.
- *Problemas de construcción*: Construir un valor que satisfaga una propiedad.
- *Problemas de optimización*: Encontrar el mejor valor, en los datos de entrada, que satisfaga una propiedad.
- *Problemas de decisión:* Decidir si una entrada satisface o no una propiedad.
- *Problemas adaptativos*: Mantener una propiedad todo el tiempo.

Si consideramos los autos de un estacionamiento como datos de entrada, un problema de búsqueda podría ser por ejemplo encontrar un auto de color blanco. Un problema de estructura es acomodar los autos en una sola fila. Un problema de optimización es encontrar el auto más caro. Un problema de decisión consiste en seleccionar un auto y contestar a la pregunta ¿el auto tiene 4 puertas?. Un problema adaptativo es tener sólo autos con una altura máxima de 1.70 metros. Si consideramos ahora un estacionamiento vacío, un problema de construcción es por ejemplo llenar el estacionamiento al 50%.

Los problemas de dificultad se casifican en:

- *Problemas conceptualmente difíciles*: No se tiene un algoritmo que resuelva el problema, ya que no es posible entender el problema.
- *Problemas analíticamente difíciles*: Se tiene un algoritmo que resuleve el problema, pero no se sabe como analizarlo ni como resolver cada ejemplar.
- *Problemas computacionalmente difíciles*: Se tiene el algoritmo, el cual se puede analizar, pero el ánalisis indica que resolver un ejemplar se toma años.
- *Problemas computacionalmente sin solución.* No se tiene un algoritmo que resuelva el problema, ya que no es factible construir tal algoritmo.

Ahora nos enfocararemos en los problemas computacionalmente difíciles.

Una medida de los datos de entrada de un problema es llamada *tamaño de la entrada.* Por ejemplo para el problema del agente viajero el número de ciudades es el tamaño de la entrada para un ejemplar dado.

Una *operación elemental* es una operación básica (suma, resta, multiplicación y división), una comparación o una iteración.

Sea n el tamaño de la entrada de un problema A. Un algoritmo es llamado *eficiente* si resuelve el problema A realizando menos de n^k operaciones elementales, para algún k entero [5].

Un problema es llamado *tratable* si existe una algoritmo eficiente que lo resuelva[5]. En otro caso el problema es llamado *intratable*. El conjunto formado por todos los problemas tratables se denota por P.

Existen problemas que aún no se han podido clasificar como tratables o intratables. Los problemas llamados NP-Completos están en tal situación. La característica que define a los problemas *NP-Completos* es que si se demuestra que uno es tratable entonces todos son tratables [5].

Para resolver problemas NP-Completos se utilizan algoritmos de aproximación y heurísticos [1]. Para problemas de optimización combinatoria dichos algoritmos no garantizan encontrar la solución óptima global.

2.1.2 El Problema del Agente Viajero

Como problema de requerimientos el problema del agente viajero es un problema de optimización. Por otra parte, como problema de dificultad el problema del agente viajero es un problema computacionalmente difícil. Mas especificamente es un problema NP-Completo [5].

Existe diversos algoritmos que permiten resolver el problema del agente viajero. Algunos de esos algoritmos son [13]:

- Búsqueda local;
- Templado simulado;
- Búsqueda tabú;
- Algoritmos genéticos;
- Programación lógica con restricciones;
- Redes de neuronas artificiales.

Las propuestas de redes de neuronas artificiales para resolver problemas de optimización combinatoria son [13]:

- Máquina de Boltzmann;
- Máquina de Cauchy;

- Máquina de Gauss.

En las siguientes secciones analizaremos los algoritmos de busqueda local, templado simulado y máquina de Boltzmann. Por una parte búsqueda local es un algoritmo de aproximación mientras que templado simulado y máquina de Boltzmann son algoritmos heurísticos. Hemos elegido estos tres algoritmos dada su fuerte relación. Templado simulado puede ser visto como una generalización de búsqueda local y la máquina de Boltzmann utiliza los principios del templado simulado.

2.2 Búsqueda Local

El algoritmo de búsqueda local es un algoritmo de aproximación diseñado para resolver cualquier problema de optimización combinatoria[1]. En esta sección primero revisaremos la técnica general de este algoritmo y después revisaremos como aplicar dicho algoritmo para resolver el problema del agente viajero.

2.2.1 Técnica General

Sea (E, S, f) un ejemplar de un problema de optimización combinatoria y $\mathcal{N}(x, \sigma)$ una estructura de vecindad. El algoritmo de *búsqueda local* consiste en:

1. Seleccionar aleatoriamente una solución $x \in S$. Esta solución es considerada la solución actual.
2. Aplicar la operación simple σ a x. Con lo anterior se genera una solución $y \in S_x$.
3. Si $f(y) < f(x)$ la solución y es considerada la solución actual.
4. Repetir los pasos 2 y 3 hasta terminar de revisar todas las soluciones $y \in S_x$ [15].

El algoritmo de búsqueda local no garantiza encontrar un óptimo global [15]. Este algoritmo termina en un óptimo local, un óptimo local de la vecindad de la solución inicial.

Una forma para intentar escapar de los óptimos locales es llevar a cabo el algoritmo de búsqueda local muchas veces, comenzando con diferentes soluciones iniciales y conservando el mejor resultado [15].

[1] Siempre y cuando sea posible definir una estructura de vecindad.

2.2.2 Búsqueda Local para el Problema del Agente Viajero

Sea $G = (V, A)$ una gráfica y $x = ij, jp, pq, ..., uv, vi$ un ciclo hamiltoniano de G. Entonces podemos abreviar dicho ciclo anotando únicamente el vértice inicial de cada arco, es decir:

$$x = i, j, p, ..., u, v.$$

Tenemos entonces que x es una permutación de V. Escribiremos a dicha permutación como un vector es decir:

$$x = [x_1, x_2, \dots, x_n]$$

donde $n = |V|$.

Considerando lo anterior redefinimos el problema del agente viajero como un problema de optimización combinatoria como sigue:

Sea $V = \{1, 2, 3, \dots, n\}$ y $D = [d_{ij}]$ una matriz $n \times n$ de distancias, el problema del agente viajero consiste en encontrar una solución mínima global de (E, S, f), donde:

- $E = \{\text{Permutaciones de } V\}$;
- $S = E$;
- $f(x) = \sum_{i=1}^{n-1} d_{i,i+1} + d_{n,1}$.

Ahora necesitamos definir una estructura de vecindad.

Sea $x = [x_1, x_2, \dots, x_n]$ una permutación de $V = \{1, 2, 3, \dots, n\}$. La *operación simple 2-cambio* $\sigma_2(q, p)$ consiste en intercambiar los valores de las posiciones q, p y de las posiciones intermedias. Más formalmente, si la solución y es obtenida a partir de x por medio de $\sigma_2(q, p)$ entonces:

$$y_i = \begin{cases} x_p & \text{si } i = q \\ x_q & \text{si } i = p \\ x_{p-1} & \text{si } i = q + 1 \wedge i < p \\ x_i & \text{en otro caso} \end{cases}$$

Ejemplo: Sea $x = [2, 4, 6, 1, 3, 5]$ entonces $y = [2, 3, 1, 6, 4, 5]$ es una solución obtenida de x por medio de la operación simple $\sigma_2(2, 5)$.

Sea (E, S, f) un ejemplar del problema del agente viajero. La *estructura de vecindad 2-cambio* $\mathcal{N}(x, \sigma_2(q, p))$ define para cada solución $x \in S$ el siguiente conjunto:

$$S_x = \{y \in S \mid y \text{ es obtenida de } x \text{ por medio de } \sigma_2(q, p)\}.$$

2.3 Templado Simulado

El algoritmo de templado simulado es un algoritmo heurístico diseñado para resolver cualquier problema de optimización combinatoria [2]. En esta sección primero revisaremos la técnica general de este algoritmo y después revisaremos como aplicar dicho algoritmo para resolver el problema del agente viajero.

2.3.1 Técnica General

La *mecánica estadística* es la disciplina, de la física de la materia condensada, que estudia las propiedades (estados) de la materia macroscópica a partir de sus constituyentes microscópicos (configuraciones átomicas) [22].

Un *sólido en estado estable* es un sólido cuya configuración átomica tiene un valor de energía mínimo [15].

Uno de los objetivos de la mecánica estadística es encontrar procesos que permitan alcanzar estados estables en sólidos [1]. Uno de tales procesos es el *templado.*

El proceso de templado consiste en:

1. Introducir el sólido en un baño caliente.
2. Incrementar la temperatura del baño caliente hasta un valor máximo en el que el sólido se funde.
3. Bajar cuidadosamente (lentamente) la temperatura del baño caliente hasta que las partículas se reacomodan en un estado estable del sólido [15].

En el proceso de templado si la temperatura máxima es suficientemente alta y el enfriado es suficientemente lento y cercano al punto de congelación el sólido alcanza un estado estable [15].

Para explicar teóricamente el proceso físico de templado la mecánica estadística utiliza la distribución de Boltzmann-Gibbs y el hecho de que la configuración átomica de un sólido puede ser representada como un vector x [22].

La distribución de *Boltzmann-Gibbs* está dada por:

$$P_T\{x = i\} = \frac{exp\left(-\frac{E_i}{K_B T}\right)}{Z(T)}$$

[2] Siempre y cuando sea posible definir una estructura de vecindad.

Donde

- x es una variable aleatoria que denota el estado actual del sólido;
- T es la temperatura;
- E_i es la energía;
- K_B es la constante de Boltzmann $(1.3806568 \times 10^{-23}\,\mathrm{J\,K^{-1}})$; y
- $Z(T)$ es llamado el factor de Boltzmann y es definido como

$$Z(T) = \sum_j exp\left(-\frac{E_j}{K_B T}\right).$$

La distribución de Boltzmann-Gibbs da la probabilidad de que el sólido esté en el estado i con energía E_i a la temperatura T [22].

En 1953 Metropolis *et al*[16] introdujeron un algoritmo que proporciona una simulación eficiente de la evolución de un sólido en un baño caliente hacia un estado estable. A continuación mostramos el modelo original del algoritmo Metropolis.

> "Calculamos entonces el cambio de energía del sistema ΔE, el cual es causado por el movimiento. Si $\Delta E < 0$, es decir el movimiento conducirá al sistema hacia un estado de baja energía, permitimos el movimiento y ponemos a la partícula en su nueva posición. Si $\Delta E > 0$, permitimos el movimiento con la probabilidad $exp\left(-\frac{\Delta E}{KT}\right)$; es decir, tomamos un número aleatorio ε entre 0 y 1, y si $\varepsilon < exp\left(-\frac{\Delta E}{KT}\right)$; movemos la partícula hacia su nueva posición. Si $\varepsilon > exp\left(-\frac{\Delta E}{KT}\right)$; la regresamos a su antigua posición." [16]

El *templado simulado* (simulated annealing) es un algoritmo que implementa el algoritmo Metropolis para resolver problemas de optimización combinatoria [15]. Este algoritmo fue introducido de forma independiente, primero por el grupo formado por Kirkpatrick, Gelatt y Vechi en 1982, y después por Cerny en 1985 [1].

El algoritmo de templado simulado utiliza la siguiente analogía entre mécanica estadística y optimización combinatoria:

1. Estados del sólido $\Longleftrightarrow$ Soluciones factibles.
2. Energía $\Longleftrightarrow$ Costo.
3. Temperatura $\Longleftrightarrow$ Parámetro de control.
4. Estado estable del sólido$\Longleftrightarrow$ Solución óptima [4][15].

El concepto de temperatura no tiene equivalente en un problema de optimización combinatoria. La temperatura es modelada con un parámetro de control, el cual tiene las mismas unidades que la función de costo [15].

Sea (E, S, f) un ejemplar de un problema de optimización combinatoria y $\mathcal{N}(x, \sigma)$ una estructura de vecindad. El algoritmo de templado simulado consiste en:

1. Asignar valores iniciales a los párametros de tiempo y control: $t = 0$; $c_t = c_0$
2. Seleccionar aleatoriamente una solución $x \in S$.
3. Aplicar la operación simple σ a x. En otras palabras, generar una solución $y \in S_x$.
4. Sea $\Delta f = f(y) - f(x)$, si $\Delta f <= 0$ entonces $x = y$.
5. En otro caso ($\Delta f > 0$), sea ε un número aleatorio entre 0 y 1. Si $\varepsilon < exp\left(\frac{-\Delta f}{c_k}\right)$ entonces $x = y$.
6. Repetir los pasos 3-5 L veces (L es llamado parámetro de longitud).
7. Incrementar el parámetro de tiempo: $t = t + 1$.
8. Bajar el parametro de control c_t.
9. Repetir los pasos 6-8 hasta que se cumpla un criterio de parada [1].

El algoritmo de templado simulado en teoría converge hacia solución óptima global [15]. Dicha convergencia esta garantizada cuando $c \to 0$ y $L \to \infty$ [1]. En la práctica este algoritmo encuentra soluciones de alta calidad, que no depende de la solución inicial [1].

Para implementaciones en computadora, una forma simple de bajar el parámetro de control es:

$$c_t = \alpha \cdot c_t$$

Donde α es un número menor y cercano a 1, por ejemplo 0.95 o 0.99 [1].

2.3.2 Templado Simulado para el Problema Agente Viajero

Para utilizar el algoritmo de templado simulado para resolver el problema del agente viajero se utiliza la definición de dicho problema como un ejemplar de un problema de optimización combinatoria dada en el algoritmo de búsqueda local y la misma estructura de vecindad $\mathcal{N}(x, \sigma_2(q, p))$ [1].

Para este algoritmo se tiene que:

$$\Delta f = d_{q-1,p} + d_{q,p+1} - d_{q-1,q} - d_{p,p+1}$$

Donde:

$$q-1=\begin{cases} q-1 & \text{si } q>1; \\ n & \text{si } q=1. \end{cases} \qquad p+1=\begin{cases} p+1 & \text{si } p<n; \\ 1 & \text{si } p=n. \end{cases}$$

La fórmula anterior permite calcular la diferencia de distancia Δf sin calcular la distancia de la solución actual y la distancia de la solución vecina (generada por la operación simple $\sigma_2(q,p)$).

2.4 Máquina de Boltzmann Secuencial

El algoritmo llamado máquina de Boltzmann es una propuesta del campo de las redes de neuronas artificiales para resolver problemas de optimización combinatoria. Dicho algoritmo es heurístico, es decir no hay garantia de encontrar una solución óptima global. En esta sección primero revisaremos la técnica general de este algoritmo y después revisaremos como aplicar dicho algoritmo para resolver el problema del agente viajero.

2.4.1 Técnica General

Las *redes de neuronas artificiales* son redes interconectadas masivamente y en paralelo de elementos simples. Ellas intentan simular el sistema nervioso biológico [14].

Un modelo de red de neuronas artificiales está constituido por:

1. Un conjunto finito de unidades de procesamiento, llamadas *neuronas artificiales*
2. Un conjunto finto de valores reales que pueden tomar las neuronas, llamado *estados de activación posibles.* El estado de activación de una neurona i al tiempo t es denotado como $a_i(t)$. El conjunto de estados de activación de todas las neuronas al tiempo t , $a(t)$, es llamado *patrón de activación.*
3. Una matriz real W , llamada *matriz de pesos de conexión.* A cada par de neuronas ij se les asocia un valor real w_{ij}, llamado *peso de conexión.* Así, la matriz W está formada por los pesos de conexión de todas las parejas de neuronas.
4. Una función que asocia a cada neurona i un número real Net_i, llamada *entrada neta de la neurona i.* La entrada neta generalmente es calculada como $Net_i = \sum_j w_{ij} \cdot a_j(t)$.
5. Una función F que asigna a cada neurona i, un nuevo estado de activación $a_i(t+1)$, llamada *función de activación.* El nuevo estado de activación es calculado como $a_i(t+1) = F(Net_i, a_i(t))$.
6. Una regla que indique como construir (y modificar) la matriz W, llamada *regla de entrenamiento* [21][14][23][9].

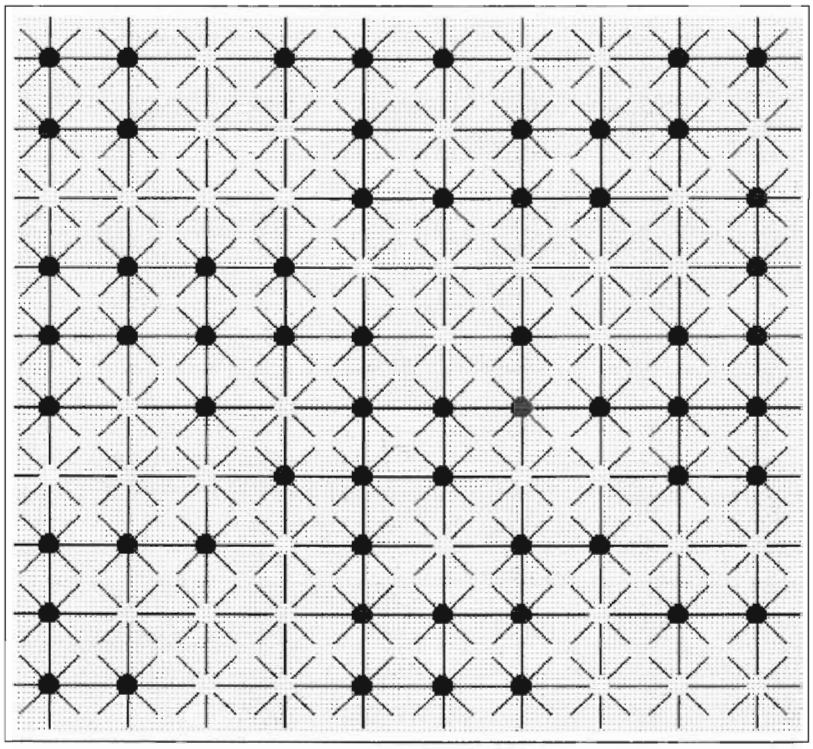


Figura 2.1: Red de neuronas artificiales

El funcionamiento de una red de neuronas artificiales puede ser descrito como sigue. La red de neuronas artificiales comienza con un patrón de activación inicial $a(t)$; es decir, a cada neurona se le asigna un estado de activación inicial. Las neuronas repetitivamente examinan sus entradas netas y deciden cambiar su estado de activación, de acuerdo a la función de activación F. Frecuentemente, la red converge a un patrón de activación estable del espacio de patrones de activación [23][21][9]. En la figura 2.1 mostramos un ejemplo de una red de neuronas artificiales.

El funcionamiento de una red de neuronas artificiales puede ser paralelo o secuencial. En el funcionamiento paralelo muchas neuronas pueden cambiar su estado de activación al mismo tiempo. En el funcionamiento secuencial en cada tiempo solamente una nerona puede cambiar su estado de activación [21].

Un ejemplo de red de neuronas artificiales es la red de Hopfield [12].

Las características de la red de Hopfiel son:

- Tiene n neuronas.
- Los estados de activación posibles son 0 y 1. Por lo que, el patrón de activación puede ser representado como un vector binario.

- La matriz de pesos W es una matriz simétrica ($w_{ij} = w_{ji}$) con ceros en la diagonal ($w_{ii} = 0$).
- La entrada neta es calculada como:

$$Net_i = \sum_j w_{ij} \cdot a_j(t).$$

- La función de activación F es definida como:

$$F(Net_i, a_i(t)) = \begin{cases} 1 & \text{si } Net_i > 0; \\ 0 & \text{si } Net_i < 0; \\ a_i(t) & \text{si } Net_i = 0. \end{cases}$$

- La regla de entrenamiento para construir la matriz de pesos W es:

$$w_{ij} = \begin{cases} \sum_j \left(2V_i^k - 1\right)\left(2V_j^k\right) & \text{si } i \neq j \\ 0 & \text{si } i = j \end{cases}$$

  Esta regla permite guardar un conjunto de recuerdos:

$$V = \left\{V^1, V^2, ..., V^s\right\}$$

  Cada recuerdo V^k debe ser representado como un patrón de activación particular, es decir como un vector binario de longuitud n.

- La matriz de pesos permanece fija durante el funcionamiento [12].

La *máquina de Boltzmann* es una red de neuronas artificiales basada en la red de Hopfield y el algoritmo de templado simulado [13] [2]. Esta red fue introducida por Hinton y Sejnowski en 1983 [1].

Las caracteristicas de la máquina de Boltzmann son :

- Tiene n neuronas.
- Los estados de activación posibles son 0 y 1. Por lo que, el patrón de activación puede ser representado como un vector binario.
- La matriz de pesos W es una matriz simétrica ($w_{ij} = w_{ji}$). A diferencia de la red de Hopfield esta matriz no tiene ceros en la diagonal ($w_{ii} \neq 0$).
- La entrada neta es calculada como:

$$Net_i = \sum_{j=1}^{n} w_{ij} \cdot a_j(t)$$

- La función de activación F es una función estocástica definida como:

$$F(Net_i, a_i(t), c_t) = \begin{cases} 1 - a_i(t) & \text{si } \left[1 + exp\left(-\frac{\Delta C(i)}{c_t}\right)\right]^{-1} > \varepsilon; \\ a_i(t) & \text{en otro caso.} \end{cases}$$

- Donde:
  - $\Delta C(i) = (1 - 2 \cdot a_i(t)) \cdot [Net_i + (1 - a_i(t)) \cdot w_{ii}]$;
  - ε es un número aleatorio del intervalo (0,1);
  - c_t es el parámetro de control al tiempo t.
- La regla de entrenamiento para construir (y modificar) la matriz de pesos W depende del problema a solucionar [21][1].

El funcionamiento de la máquina de Boltzmann puede ser paralelo o secuencial. La máquina de Boltzmann es llamada *máquina de Boltzmann secuencial* si tiene un funcionamiento secuencial [1].

La máquina de Boltzmann secuencial está descrita por el siguiente algoritmo:

1. Asignar valores iniciales a los parametros de tiempo y control: $t = 0$; $c_t = c_0$.
2. Generar aleatoriamente un patron de activación inicial. En otras palabras, asignar a cada neurona un estado de activación inicial generado aleatoriamente en $\{0, 1\}$.
3. Seleccionar aleatoriamente una neurona i y determinar su nuevo estado de activación, de acuerdo a la función de activación F, es decir:
$$a_i(t) = F(Net_i, a_i(t), c_t)$$
4. Repetir el paso 3 n veces.
5. Incrementar el tiempo: $t = t + 1$
6. Bajar el parámetro de control c_t:
$$c_t = \frac{c_0}{(1 + t)}.$$
7. Repetir los pasos 3-6 hasta que la red converja [21][10].

Para una implementación en computadora de la máquina de Boltzmann secuencial se considera el siguiente *criterio de convergencia*. La red converge si durante K decrementos consecutivos del parámetro de control no se registran cambios en los estados de activación de las neuronas [1]. Llamaremos a K el *parámetro de convergencia*. El valor del parámetro de convergencia es tomado como:

$$K = 10.$$

A continuación mostraremos algunos aspectos teóricos de la máquina de Boltzmann que nos serán de utilidad en las implementaciones.

El *espacio de patrones de activación* $\mathcal{A}$ de una máquina de Boltzmann es definido como el conjunto de todas los patrones de activación posibles. Es claro que la cardinalidad de $\mathcal{A}$ es igual a 2^n [1].

Sea $a \in \mathcal{A}$, entonces la función de consenso de la máquina de Boltzmann es definida como:

$$C : \mathcal{A} \longrightarrow R$$
$$C(a) = \sum_i \sum_j w_{ij} \cdot a_i \cdot a_j \quad i, j = 1, .., n.$$

La función de consenso asigna a cada patrón de activación a un numero real $C(a)$, llamado *consenso* [1].

Sea $a \in \mathcal{A}$, entonces a es llamado *máximo global* si:

$$C(a) \geq C(b) \quad \text{para toda } b \in \mathcal{A}$$

Sea $a \in \mathcal{A}$, entonces un *patrón de activación vecino* a^i de a es definido como el patrón de activación que es obtenido de a al cambiar el estado de la neurona i [1]. Así se tiene que:

$$a_j^i = \begin{cases} a_j & \text{si } j \neq i; \\ 1 - a_j & \text{si } j = i. \end{cases}$$

La *diferencia de consenso* entre dos configuraciones vecinas a y a^i es calculada como:

$$\Delta C(i) = C(a^i) - C(a).$$

La diferencia de consenso puede ser calculada como:

$$\Delta C(i) = (1 - 2 \cdot a_i) \cdot [Net_i + (1 - a_i) \cdot w_{ii}].$$

De esta forma la diferencia de consenso es un cálculo local y es realizada por cada neurona [1].

Sea $a \in \mathcal{A}$, entonces a es llamado *máximo local* si:

$$\Delta C(i) \leq 0 \quad \text{para toda } i.$$

En otras palabras un máximo local es un patrón de activación cuyo consenso no puede incrementarse con un cambio de estado de una sola neurona [1].

En teoría la máquina de Boltzmann converge hacia un patrón de activación máximo global [1].

2.4.2 Máquina Boltzmann Secuencial para el Problema del Agente Viajero

Dada una matriz $n_1 \times n_1$ de distancias, $D = [d_{ij}]$, la máquina de *Boltzmann para el problema del agente viajero* es definida por:

- $n = n_1 \times n_1$ neuronas. Las neuronas se enumeran comenzando en 00 y terminando en $(n_1 - 1)(n_1 - 1)$. Así un patron de activación es:

$$a = [a_{00}, a_{01}, \ldots a_{(n_1-1)(n_1-1)}]$$

- La matriz de pesos W es una matriz $n \times n$ dada por:

$$w_{(ip)(jq)} = \begin{cases} max\{d_{ik} + d_{il}\} + I & \text{si se cumple A;} \\ d_{ij} & \text{si se cumple B;} \\ -\left(min\{w_{(ip)(ip)}, w_{(jq)(jq)}\} + I\right) & \text{si se cumple C;} \\ 0 & \text{en otro caso.} \end{cases}$$

- Donde:
  - A: $(i = j) \wedge (p = q)$;
  - B: $(i \neq j) \wedge (q = p + 1 \ mod \ n_1)$;
  - C: $(i = j \ \wedge \ q \neq p) \vee (i \neq j \wedge q = p)$;
  - I es un número real positivo.
- La matriz de pesos W permanece fija durante el funcionamiento [1].

Para interpretar un patrón de activación como una solución del problema del agente viajero, este se organiza como un matriz de $n_1 \times n_1$, es decir:

$$a(t) = \begin{vmatrix} a_{00} & a_{01} & \cdots & a_{0(n_1-1)} \\ a_{10} & a_{11} & \cdots & a_{1(n_1-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n_1-1)0} & a_{(n_1-1)1} & \cdots & a_{(n_1-1)(n_1-1)} \end{vmatrix}$$

Así los patrones de activación corresponden a soluciones del problema del agente viajero formulado como un problema de programación lineal 01. Por lo que un patrón de activación es una solución factible si:

$$\sum_i a_{ip} = 1, \qquad p = 0, \ldots, (n_1 - 1);$$

$$\sum_p a_{ip} = 1, \qquad i = 0, \ldots, (n_1 - 1).$$

Es decir, dicha matriz binaria tiene únicamente una entrada con valor 1 en cada renglón y en cada columna. Lo anterior se interpreta como las condiciones de que una ciudad no puede ser visitada más de una vez y se deben visitar todas las ciudades.

En teoría la máquina de Boltzmann converje hacia un patrón de activación que corresponde a una solución mínima global del problema del agente viajero [1].

La distancia del viaje puede ser calculada como:

$$d = Const - C(a)$$

Donde:

- $Const$ es una constante que depende del problema;
- $C(a)$ es el consenso del patrón de activación a [1].

Capítulo 3

Algoritmos Modificados

En este capítulo presentamos nuestras porpuestas de modificación para los algoritmos siguientes:

- Búsqueda local
- Templado simulado
- Máquina de Boltzmann secuencial.

Las modificaciones realizadas a los algoritmos están enfocadas a mejorar la calidad de las soluciones encontradas. Más específicamente, modificarenos los algoritmos con el objetivo de encontrar distancias menores para el problema del agente viajero.

3.1 Búsqueda Local Modificada

Las modificaciones que proponemos para este algoritmo son:

- Realizar cambio de vecindad. Es decir, una vez encontrada una solución y, mejor que la solución actual, consideraremos a S_y como la vecindad actual.
- Generar las soluciones vecinas $y \in S_x$ de manera aleatoria.
- Utilizar un criterio convergencia. Pararemos el algoritmo si durante K aplicaciones consecutivas de la operación simple σ no se encuentra una solución mejor que la solución actual.
- Utilizar la diferencia de costo Δf del templado simulado para el problema del agente viajero.
- Aceptar soluciones con el mismo costo que la solución actual, es decir $\Delta f = 0$

Con los cambios anteriores se exploran más soluciones que en el algoritmo original. Por ello esperamos encontrar mejores soluciones para el problema del agente viajero.

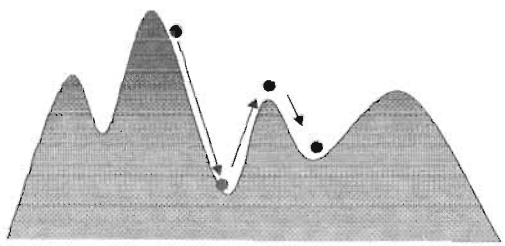
A continuación presentamos el algoritmo modificado.

Sea (E, S, f) un ejemplar de un problema de optimización combinatoria y $\mathcal{N}(x, \sigma)$ una estructura de vecindad. El algoritmo de *búsqueda local modificado* consiste en:

1. Seleccionar aleatoriamente una solución $x \in S$. Esta solución es considerada la solución actual.
2. Asignar valor inicial al parámetro de convergencia: $conv = 0$.
3. Asignar valor inicial al parámetro de cambio: $cambio = 0$.
4. Aplicar la operación simple σ a x de manera aleatoria. En otras palabras, generar aleatoriamente una solución $y \in S_x$.
5. Sea $\Delta f = f(y) - f(x)$, si $\Delta f <= 0$ entonces $x = y$; si $\Delta f < 0$ entonces $cambio = 1$.
6. Si $cambio = 0$ entonces $conv = conv + 1$; en otro caso $conv = 0$.
7. Repetir los pasos 3-6 mientras que $conv < K$.

3.2 Templado Simulado Modificado

Las modificaciones para este algoritmo están basadas en la suposición de que en la práctica el algoritmo encuentra soluciones mejores que la solución a la que converje (solución final). Dado que el algoritmo escapa de los óptimos locales (cuando $c_t > 0$) es posible que la solución final no sea mejor que alguna solución anterior. La siguiente figura ilustra de manera simplificada nuestra suposición:



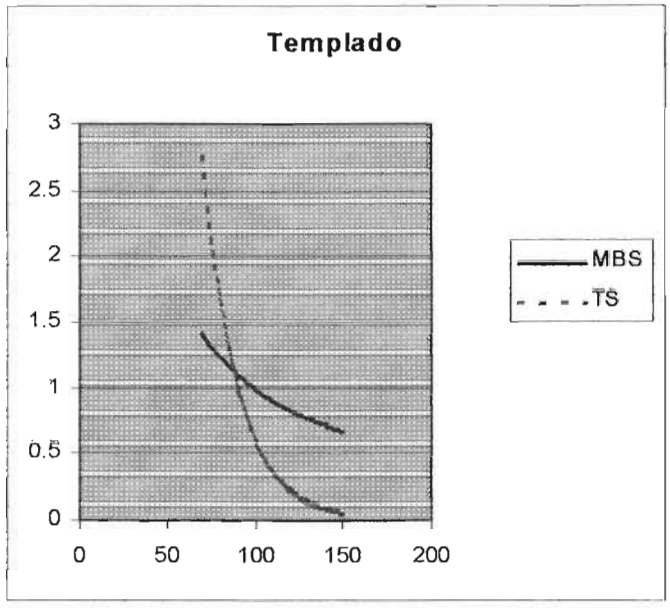
Las modificaciones que proponemos para este algoritmo son las siguientes:

- Utilizar el templado de la máquina de Boltzmann secuencial, es decir:

$$c_t = \frac{c_0}{1+t}$$

- Conservar la mejor solución.

Cambiamos el templado ya que el de la máquina de Boltzmann secuencial es mucho más lento, como se puede apreciar en la siguiente figura:



Como se puede observar, para valores mayores de 100 del parámetro de tiempo, el templado de la máquina de Boltzmann (línea continua) produce valores más grandes para el parámetro de control que los del templado simulado (línea punteada). Con lo anterior tenemos que con este cambio nuestro algoritmo realizará más comparaciones que el algoritmo original.

Con el cambio de templado se exploran más soluciones y guardando la mejor solución encontrada no se pierde dicha solución. Por lo anterior esperamos encontrar mejores soluciones, que las del algoritmo original, para el problema del agente viajero.

A continuación presentamos el algoritmo modificado.

Sea (E, S, f) un ejemplar de un problema de optimización combinatoria y $\mathcal{N}(x, \sigma)$ una estructura de vecindad. El algoritmo de *templado simulado modificado* consiste en:

1. Asignar valores iniciales a los parametros de tiempo y control: $t = 0$; $c_t = c_0$.
2. Seleccionar aleatoriamente una solución $x \in S$.
3. Guardar la solución actual: $x_{min} = x$; $f_{min} = f(x)$.
4. Aplicar la operación simple σ a x. En otras palabras generar una solución $y \in S_x$.

5. Sea $\Delta f = f(y) - f(x)$, si $\Delta\ f <= 0$ entonces $x = y$; si $f(x) < f_{min}$ entonces guardar la solución actual: $x_{min} = x$; $f_{min} = f(x)$.

6. En otro caso ($\Delta\ f > 0$), sea ε un número aleatorio entre 0 y 1. Si $\varepsilon < exp\left(\frac{-\Delta f}{c_k}\right)$ entonces $x = y$.

7. Repetir los pasos 4-6 L veces.

8. Incrementar el tiempo: $t = t + 1$

9. Bajar el parametro de control:

$$c_t = \frac{c_0}{1+t}$$

10. Repetir los pasos 7-9 hasta que se cumpla un criterio de parada.

11. Mostrar la solución x_{min}.

Tanto para el algoritmo original como para el modificado consideraremos:

- Generación de soluciones vecinas $y \in S_x$ de manera aleatoria.
- El parámetro $L = n$ (número de ciudades).
- Un criterio de parada similar al criterio de convergencia de la máquina de Boltzmann secuencial. Es decir, pararemos el algoritmo si durante K decrementos consecutivos del parámetro de control no se aceptan soluciones con un costo menor o mayor que el costo de la solución actual x.

3.3 Máquina de Boltzmann Secuencial Modificada

Al igual que en el algoritmo templado simulado, las modificaciones para este algoritmo están basadas en la suposición de que en la práctica el algoritmo encuentra soluciones mejores que la solución a la que converje.

Las modificaciones que proponemos para este algoritmo son:

- Trabajar con la distancia. Para ello utilizaremos la fórmula dada anteriormente en la sección 2.4.2:

$$d = Const - C(a)$$

- Conservar el mejor patrón de activación (solución).

Queremos trabajar con la distancia para tener la posibilidad de ir mostrando la distancia conforme se corre el algoritmo. Al guardar la mejor solución (patrón de activación) no se nos pierde y también podemos mostrarla durante la corrida del algoritmo. De esta forma, aunque el algoritmo converja a una solución con mayor distancia el algoritmo nos regresará la solución con menor distancia encontrada.

Considerando lo anterior, mostramos el algoritmo modificado.

La máquina de *Boltzmann secuencial modificada* está descrita por:

1. Asignar valores iniciales a los parametros de tiempo y control: $t = 0$; $c_t = c_0$.
2. Generar aleatoriamente un patrón de activación inicial $a(t)$. En otras palabras, asignar a cada neurona un estado de activación inicial generado aleatoriamente en $\{0, 1\}$.
3. Calcular la distancia actual: $d = Const - C(a(t))$.
4. Guardar la solución actual: $a_{min} = a(t)$; $d_{min} = d$.
5. Seleccionar aleatoriamente una neurona i y determinar su nuevo estado de activación, de acuerdo a la función de activación F, es decir:

   Si $\left[1 + exp\left(-\frac{\Delta C(i)}{c_t}\right)\right]^{-1} > \varepsilon$ entonces

   $a_i(t) = 1 - a_i(t)$ y calcular la distancia actual: $d = Const - C(a(t))$.

   Si $d < d_{min}$ entonces

   guardar la solución actual: $a_{min} = a(t)$; $d_{min} = d$.
6. Repetir el paso 5 n veces.
7. Incrementar el tiempo: $t = t + 1$.
8. Bajar el parámetro de control c_t :

$$c_t = \frac{c_0}{(1+t)}.$$

9. Repetir los pasos 5-8 hasta que la red converja.
10. Mostrar el patrón de activación a_{min}.

Para calcular *Const* primero se tiene que correr el algoritmo original; después calcular $C(a)$, acomodar a como una solución x (matriz) y calcular su distancia $f(x)$ con la fórmula dada en sección 1.2.2.

Capítulo 4

Viaje Redondo por 51 Ciudades de México

En este capítulo probaremos los algoritmos revisados en un problema real. Además compararemos nuestras propuestas de modificación con los algoritmos originales. La comparación la realizaremos considerando la calidad de las soluciones encontradas y el número de operaciones elementales realizadas.

4.1 Definición del Problema

Deseamos conocer la distancia mínima de un recorrido por 51 ciudades de México, con las siguientes características:

- Viaje redondo. El recorrido debe de comenzar y terminar en la misma ciudad.
- Completo. El recorrido debe de pasar por todas las ciudades.
- Sin repeticiones. El recorrido debe de pasar por cada ciudad una sola vez.
- En automóvil. Para el recorrido se considerarán carreteras y autopistas.

Las ciudades que consideraremos las mostramos en el Apéndice A.

4.2 Analisis del Problema

Nuestro problema a resolver es un ejemplo concreto del problema del agente viajero. El recorrido solicitado en el Subcapítulo anterior corresponde a un ciclo hamiltoniano de longuitud mínima. La distancia de dicho recorrido es la longuitud del ciclo hamiltoniano.

Los datos de entrada son las distancias, en kilometros, entre cada par de ciudades. Dichas distancias se presentan en el Apéndice B. Los datos de salida serán un ciclo hamiltoniano con la menor longitud encontrada (solución) y su longuitud (distancia).

Las técnicas de solución que emplearemos para resolver nuestro problema son:

- Búsqueda local;
- Búsqueda local modificada;
- Templado simulado;
- Templado simulado modificado;
- Máquina de Boltzmann secuencial;
- Máquina de Boltzmann secuencial modificada.

4.3 Algoritmos Originales vs Modificados

En este Subcapítulo comenzaremos realizando pruebas de los seis algoritmos. Después compararemos los algoritmos originales con los algoritmos modificados realizando 100 pruebas para cada uno de ellos.

4.3.1 Pruebas

La codificación de los programas la realizamos en lenguaje C; los códigos fuentes aparecen en el Apéndice C. Las pruebas fueron realizadas en una PC a 600 Mz. y en ambiente Linux.

Iniciaremos con una corrida del algortimo de búsqueda local. La siguiente figura muestra la compilación y la salida[1]:

[1] En Linux el alchivo a.out es el ejecutable del último archivo compilado en lenguaje C.

La solución anterior se interpreta de la sigiente forma:

El recorrido comienza en la ciudad 24 (Mérida); de la ciudad 24 ir a la ciudad 45 (Toluca); de la ciudad 45 ir a la ciudad 1 (Acapulco); de la ciudad 1 ir a la ciudad 35 (Querétaro); etc. La distancia en km del recorrido anterior es de 67,311.

Ahora correremos nuevamente el algoritmo de búsqueda local pero 100,000 veces. En cada corrida se genera una solución aleatoria diferente. La mejor solución encontrada en esta prueba tuvo una distancia de **49,644 km.**

Realizaremos ahora una prueba del algoritmo de búsqueda local modificada. Para esta prueba tomaremos $K = 10N^2$. El resultado de esta prueba se muestra en la siguiente figura:

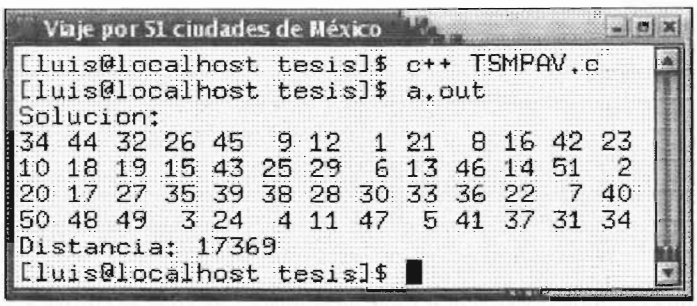
Como puede observarse la distancia obtenida (17,708) es mucho menor que la encontrada al correr 100,000 veces el algoritmo original (49,644).

Al correr búsqueda local modificada 10,000 veces la distancia miníma encontrada fue **17,369 km**. La solución correspondiente a la distancia anterior es:

26 32 44 34 31 37 41 5 47 11 4 24 3
49 48 50 40 7 22 36 33 30 28 38 39 35
27 17 20 2 51 14 46 13 6 29 25 15 19
43 18 10 23 42 16 8 21 1 12 9 45 26

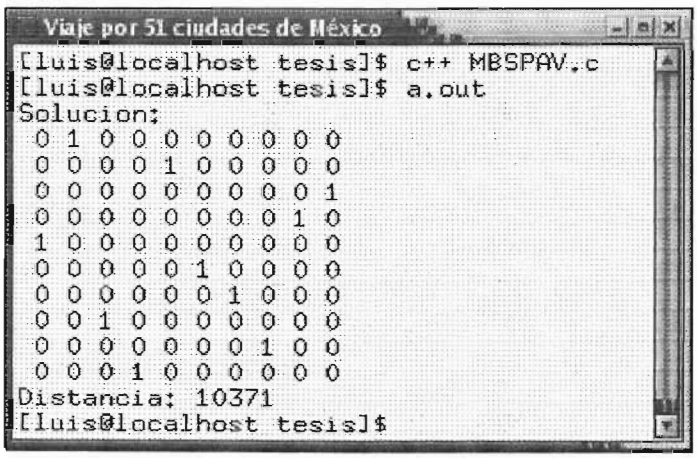
Probaremos ahora el algoritmo de templado simulado. Al realizar una corrida de templado simulado con $c_0 = 5,000,000$ y $K = 10N$ obtenemos:

Con los mismos parámetros correremos el algoritmo de templado modificado. Para este algoritmo el resultado es:



Como podemos ver la distancia encontrada por el algoritmo original (17,769) es un poco mayor que la del algoritmo modificado (17,369). Por otra parte, la distancia del algoritmo modificado es la misma que la obtenida para las 10,000 corridas de busqueda local modificada.

Realizaremos ahora pruebas para el algoritmo máquina de Boltzmann secuencial. Dado que este algoritmo trabaja soluciones no factibles y con tamaño cuadrático con respecto al número de ciudades empezaremos las pruebas con las primeras 10 ciudades. El resultado obtenido con máquina de Boltzmann Secuencial para $n_1 = 10$ (10 ciudades), $c_0 = 10,000,000$ y $K = 10$ es:



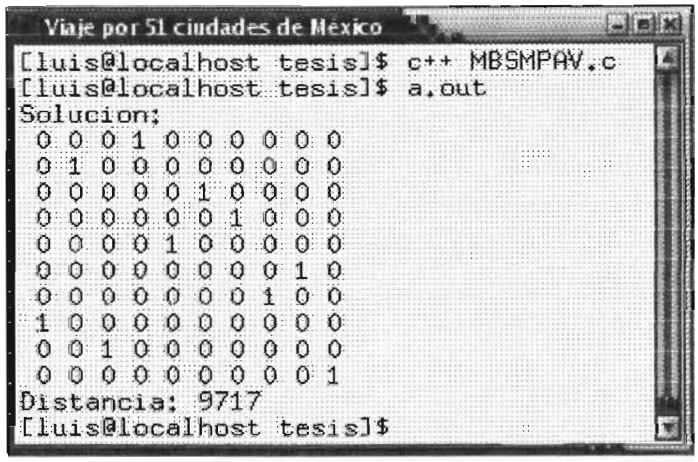
La solución anterior se interpreta como sigue:

Los renglones representan a las ciudades y las columnas a su posición en el recorrido. Así el recorrido comienza en la ciudad 5 (Ciudad Cuauhtemoc); de la ciudad 5 ir a la ciudad 1 (Acapulco); de la ciudad 1 ir a la ciudad 8 (Colima); etc.

El consenso correspondiente a la solución anterior es 39,254. Entonces para este problema tenemos:

$$Const = Consenso + d = 39,254 + 10,371 = 49,625.$$
$$\Rightarrow d = 49,625 - Consenso$$

Utilizando el resultado anterior, con el algoritmo máquina de Boltzmann secuencial modificada para las primeras 10 ciudades obtenemos:



Como puede observarse la distancia de la solución encontrada, para 10 ciudades, por el algoritmo modificado (9,717) es menor que la del algoritmo original (10,371).

Finalmente, al realizar una prueba considerando las 51 ciudades ($Const = 387,398$) y tomando $c_0 = 50,000,000$ obtuvimos una distancia mínima de **37,628 km**. El tiempo de procesamiento para el resultado anterior fue de un poco más de **5 horas**. La distancia anterior es mucho mayor que la mejor distancia obtenida (17,369) por los algoritmos anteriores (búsqueda local modificada y templado simulado modificado). Por otra parte, el tiempo de procesamiento es muy alto comparado con los otros algoritmos.

4.3.2 Error Relativo y Orden de Complejidad

Calcularemos el error relativo de los algoritmos al realizar 100 corridas. Para ello tenemos:

$$E_r = \frac{d_{prom} - d_{min}}{d_{min}}.$$

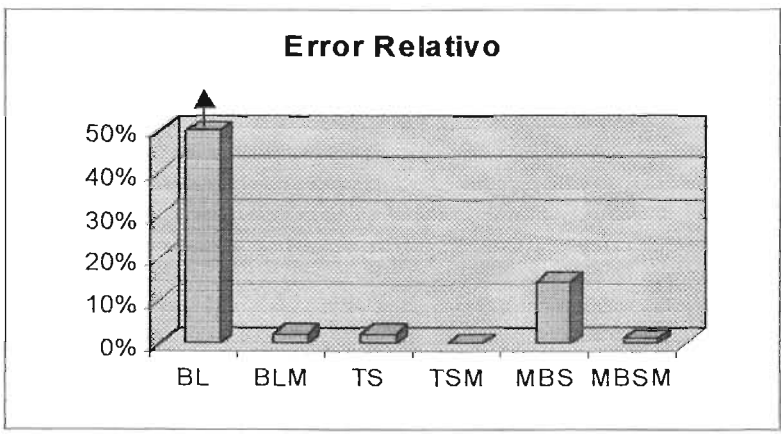
Donde:

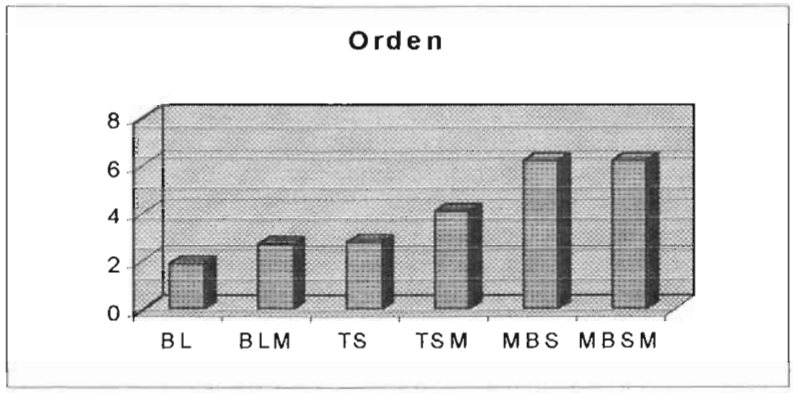
- d_{prom} es la distancia promedio para las 100 corridas.
- d_{min} es la distancia mínima encontrada por los algoritmos. Para las 51 ciudades $d_{min} = 17,369$ y para las 10 ciudades $d_{min} = 9,717$.

Para determinar el orden de complejidad de los algoritmos calcularemos el número de operaciones elementales promedio que realizan $\mathbf{ope}_{prom}$. La siguiente tabla muestra los resultados obtenidos.

Algoritmo	$\mathbf{d}_{prom}$	**Error**	$\mathbf{ope}_{prom}$	**Orden**
BL	68,304.80	293.26%	1,225.00	$n^{1.9}$
BLM	17,783.16	2.38%	33,983.22	$n^{2.7}$
TS	17,783.38	2.39%	43,816.65	$n^{2.8}$
TSM	17,369.10	0.00%	7,517,852.37	$n^{4.1}$
MBS	11,137.20	14.62%	1,380,372.00	$n^{6.2}$
MBSM	9,828.68	1.15%	1,315,156.00	$n^{6.2}$

Las siguientes gráficas muestran el error relativo y el orden de los algoritmos respectivamente.





De las gráficas anteriores tenemos las siguientes observaciones:

- Los algoritmos modificados tienen errores relativos menores que los algoritmos originales.
- El algoritmo que tiene el error relativo menor es TSM.
- El algoritmo que tiene el error relativo mayor es BL.
- Los algoritmos de BL y TS modificados tienen un orden de complejidad un poco mayor que los originales.
- Los algoritmos MBS y MBSM tienen prácticamente el mismo orden de complejidad.
- El algoritmo que tiene el menor orden de complejidad es BL.
- Los algoritmos que tienen el mayor orden de complejidad son MBS y MBSM.

Considerando lo anterior, tenemos que el mejor algoritmo (al menos para nuestro problema) es templado simulado modificado (TSM).

4.4 Solución

La solución con menor distancia (17,369) encontrada por los algoritmos búsqueda local modificada (10,000 corridas) y templado simulado modificado para el problema viaje redondo por 51 ciudades de México es[2]:

26	32	44	34	31	37	41	5	47	11	4	24	3
49	48	50	40	7	22	36	33	30	28	38	39	35
27	17	20	2	51	14	46	13	6	29	25	15	19
43	18	10	23	42	16	8	21	1	12	9	45	26

[2]Las mejores soluciones encontradas por los dos algoritmos tienen la misma distancia pero no son iguales. Aquí presentamos la mejor solución obtenida por búsqueda local modificada.

La solución anterior se interpreta como sigue:

El recorrido comienza en la ciudad 26 (México)[3]; de la ciudad 26 ir a la ciudad 32 (Pachuca); de la ciudad 32 ir a la ciudad 44 (Tlaxcala); etc.

Así, el recorrido en forma de lista (posición-ciudad) es[4]:

1. México (Distrito Federal)
2. Pachuca (Hidalgo)
3. Tlaxcala (Tlaxcala)
4. Puebla (Puebla)
5. Oaxaca (Oaxaca)
6. Salina Cruz (Oaxaca)
7. Tapachula (Chiapas)
8. Ciudad Cuauhtemoc (Chiapas)
9. Tuxtla Gutierrez (Chiapas)
10. Chetumal (Quintana Roo)
11. Cancun (Quintana Roo)
12. Mérida (Yucatán)
13. Campeche (Campeche)
14. Villahermosa (Tabasco)
15. Veracruz (Veracruz)
16. Xalapa (Veracruz)
17. Tampico (Tamaulipas)
18. Ciudad Victoria (Tamaulipas)
19. Matamoros (Tamaulipas)
20. Reynosa (Tamaulipas)
21. Piedras Negras (Coahuila)
22. Nuevo Laredo (Tamaulipas)
23. Monterrey (Nuevo León)

[3] El recorrido puede iniciar en cualquier ciudad, dado que es un viaje redondo.
[4] Esta lista representa el intinerario del viaje.

24. Saltillo (Coahuila)
25. San Luis Potosí (San Luis Potosí)
26. Querétaro (Queretaro)
27. Morelia (Michoacan)
28. Guanajuato (Guanajuato)
29. León (Guanajuato)
30. Aguascalientes (Aguascalientes)
31. Zacatecas (Zacatecas)
32. Durango (Durango)
33. Torreón (Coahuila)
34. Chihuahua (Chihuahua)
35. Ciudad Juárez (Chihuahua)
36. Nogales (Sonora)
37. Mexicali (Baja California)
38. Ensenada (Baja California)
39. La Paz (Baja California)
40. Tijuana (Baja California)
41. Hermosillo (Sonora)
42. Culiacán (Sinaloa)
43. Mazatlán (Sinaloa)
44. Tepic (Nayarit)
45. Guadalajara (Jalisco)
46. Colima (Colima)
47. Manzanillo (Colima)
48. Acapulco (Guerrero)
49. Chilpancingo (Guerrero)
50. Cuernavaca (Morelos)
51. Toluca (Estado de México)
52. México (Distrito Federal)

En la figura 4.1 mostramos el recorrido de manera gráfica.

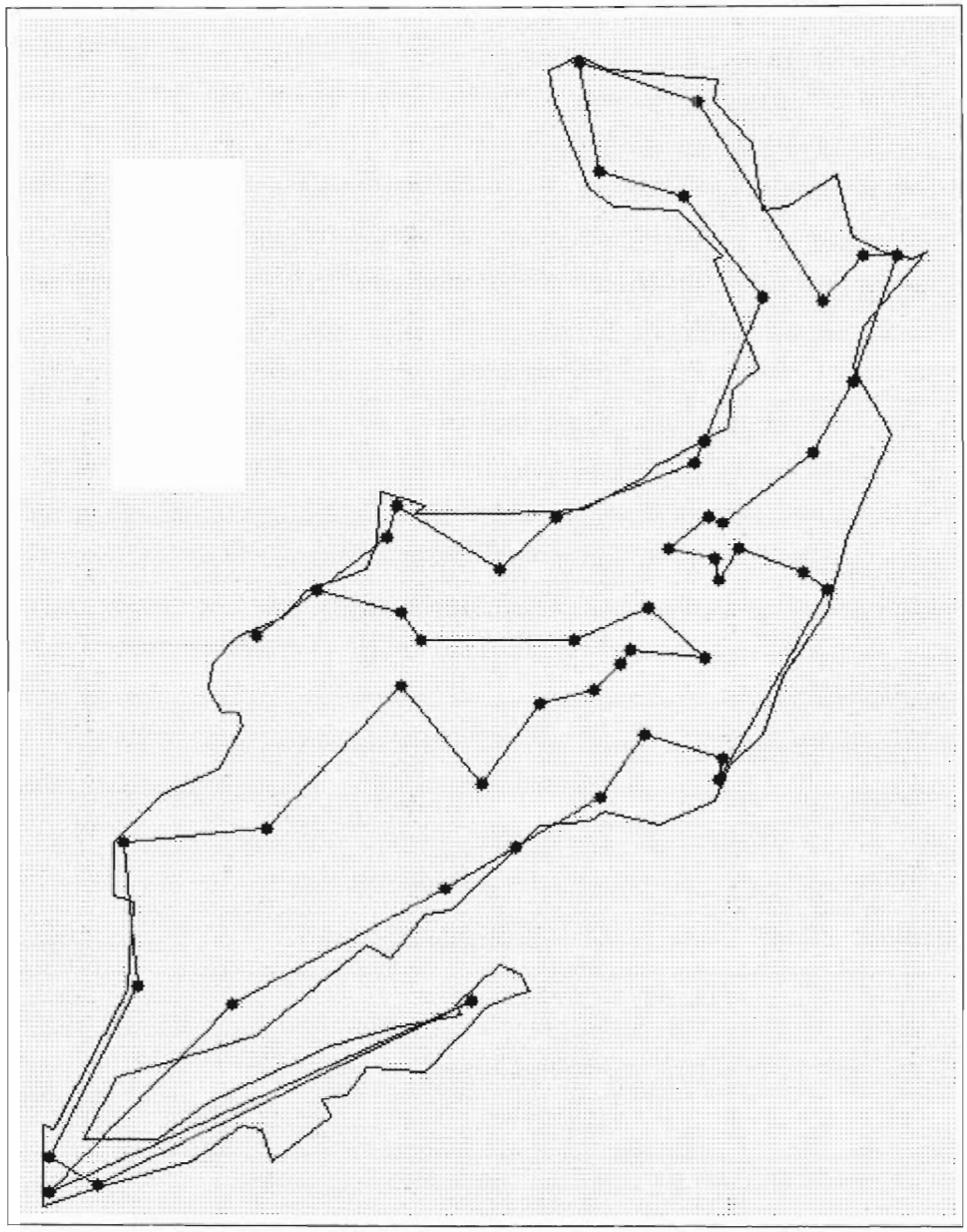


Figura 4.1: Solución para el problema viaje redondo por 51 ciudades de México.

4.5 ¿Solución Óptima Global?

El comprobar formalmente que la mejor solución encontrada para el problema viaje redondo por 51 ciudades de México ($d = 17,369$) es una solución óptima global queda fuera de los alcances de este trabajo. Sin embargo mostraremos algunos indicios que nos sugieren considerar a tal solución como una solución de muy alta calidad.

Primero, consideraremos el siguiente teorema:

Sea $G = (V, A)$ una gráfica ponderada, d_{opt} la longitud un ciclo hamiltoniano de longitud mínima y c_{opt} el peso de un arbol generardor de peso minimo. Entonces $c_{opt} < d_{opt}$ [6].

Por lo que, para encontar una cota mínima para el problema del agente viajero necesitamos encontrar un árbol generador de peso mínimo. Los algoritmos clásicos para encontrar tal arbol son Prim y Kruskal [6].

Para el problema viaje redondo por 51 ciudades de México obtuvimos $c_{opt} = 12,376$. Así tenemos que:

$$12,376 < d_{opt} \leq 17,369$$

Por otra parte, mandamos nuestros datos de entrada al sitio de internet NEOS Server for Otimization [5] y solicitamos la solución para nuestro problema con tres algoritmos diferentes (Concorde-QSopt, Concorde-GLPK,Concorde-Lin-Kernighan). Las tres soluciones que nos mandaron como respuesta coinciden con nuestra distancia mínima ($d = 17,369$) y ellos aseguran que son óptimas globales.

[5] www-neos.mcs.anl.gov

Capítulo 5

Conclusiones

El problema del agente viajero consiste en realizar un viaje redondo por n ciudades, de tal manera que la distancia total del recorrido sea la más pequeña posible.

El problema del agente viajero es un problema de optimización combinatoria. Resolver un problema de optimización combinatoria consiste en encontrar una solución óptima global. Para el problema del agente viajero se requiere econtrar una solución mínima global.

Entre los algoritmos para resolver problemas de optimización combinatoria tenemos búsqueda local, templado simulado y máquina de Boltzmann.

El algoritmo de búsqueda local da como resultado una solución óptima local. Para mejorar la calidad de la solución es necesario ejecutar este algoritmo muchas veces.

El algoritmo de templado simulado en teoría converge hacia una solución óptima global.

La máquina de Boltzmann secuencial en teoría converge hacia un patrón de activación máximo global de la función de consenso. Dicho patrón corresponde a una solución mínima global del problema del agente viajero, cuando la máquina de Boltzmann es entrenada (construcción de la matriz de pesos) para resolver este problema.

El problema viaje redondo por 51 ciudades de México es un ejemplo concreto del problema del agente viajero.

Nuestras propuestras de modificación para los algoritmos búsqueda local, templado simulado y máquina de Boltzmann secuencial tienen un mejor desempeño que los algoritmos originales. Para los algoritmos búsqueda local modificada y templado simulado modificado obtuvimos errores relativos menores con un orden de complejidad solo un poco mayor, con respecto a los algoritmos originales. Para el algoritmo máquina de Boltzmann secuencial modificada obtuvimos un error relativo menor y un orden de complejidad un poco menor, con respecto al original.

Con base en los resultados obtenidos consideramos que el mejor algoritmo (al menos para nuestro problema) es nuestra propuesta de modificación para templado simulado.

La mejor solución encontrada por los algoritmos (búsqueda local modificada y templado simulado modificado) para el problema viaje redondo por 51 ciudades de México tiene como distancia 17,369 km.

Apéndice A

Lista de Ciudades

1. Acapulco (Guerrero)
2. Aguascalientes (Aguascalientes)
3. Campeche (Campeche)
4. Cancún (Quintana Roo)
5. Ciudad Cuauhtemoc (Chiapas)
6. Ciudad Juárez (Chihuahua)
7. Ciudad Victoria (Tamaulipas)
8. Colima (Colima)
9. Cuernavaca (Morelos)
10. Culiacán (Sinaloa)
11. Chetumal (Quintana Roo)
12. Chilpancingo (Guerrero)
13. Chihuahua (Chihuahua)
14. Durango (Durango)
15. Ensenada (Baja California)
16. Guadalajara (Jalisco)
17. Guanajuato (Guanajuato)
18. Hermosillo (Sonora)

19. La Paz (Baja California)
20. León (Guanajuato)
21. Manzanillo (Colima)
22. Matamoros (Tamaulipas)
23. Mazatlan (Sinaloa)
24. Mérida (Yucatán)
25. Mexicali (Baja California)
26. México (Distrito Federal)
27. Morelia (Michoacan)
28. Monterrey (Nuevo León)
29. Nogales (Sonora)
30. Nuevo Laredo (Tamaulipas)
31. Oaxaca (Oaxaca)
32. Pachuca (Hidalgo)
33. Piedras Negras (Coahuila)
34. Puebla (Puebla)
35. Querétaro (Queretaro)
36. Reynosa (Tamaulipas)
37. Salina Cruz (Oaxaca)
38. Saltillo (Coahuila)
39. San Luis Potosí (San Luis Potosí)
40. Tampico (Tamaulipas)
41. Tapachula (Chiapas)
42. Tepic (Nayarit)
43. Tijuana (Baja California)
44. Tlaxcala (Tlaxcala)
45. Toluca (Estado de México)
46. Torreón (Coahuila)

47. Tuxtla Gutiérrez (Chiapas)
48. Veracruz (Veracruz)
49. Villahermosa (Tabasco)
50. Xalapa (Veracruz)
51. Zacatecas (Zacatecas)

Apéndice B

Tabla de distancias

En este apédice mostramos la tabla de distancias entre cada par de ciudades[1]. El número en negrita indica el número de la ciudad dado en la lista de ciudades del Apéudice A.

1										
2	908									
3	1513	1668								
4	2007	2162	494							
5	1193	1782	721	1215						
6	2258	1389	3020	3512	3132					
7	1117	515	1578	2072	1666	1493				
8	678	448	1899	2393	2013	1780	886			
9	306	602	1207	1701	1293	1952	810	833		
10	1651	970	2417	2911	2531	1466	1373	922	1351	
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

11	1703	1858	424	388	839	3208	1768	2045	1397	2607
12	117	791	1396	1890	1310	2141	999	804	189	1540
13	1882	1013	2642	3136	2756	376	1117	1404	1576	1191
14	1310	441	2070	2564	2184	1038	844	832	1004	529
15	3350	2709	4116	4610	4230	1425	2813	2621	3050	1699
16	889	246	1697	2191	1811	1578	684	202	631	720
17	760	181	1520	2014	1634	1570	559	479	454	997
18	2348	1707	3114	3608	3228	769	1811	1619	2048	697
19	4701	4060	5467	5961	5581	2776	4164	3972	4401	3050
20	782	127	1542	2036	1656	1516	541	425	476	943
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

[1] Guia Roji, 2004

11										
12	1586									
13	2832	1765								
14	2260	1193	662							
15	4306	3239	1696	2228						
16	1887	820	1202	630	2419					
17	1710	643	1194	622	2696	277				
18	3304	2237	694	1226	1002	1417	1694			
19	5657	4590	3047	3579	1351	3770	4047	2353		
20	1732	665	1140	568	2642	223	54	1640	3993	
	11	12	13	14	15	16	17	18	19	20

21	697	546	1997	2491	1890	1878	984	98	931	952
22	1370	928	1842	2336	1930	1530	317	1319	1064	1463
23	1431	750	2197	2891	2311	1347	1153	702	1131	220
24	1690	1845	177	317	898	3195	1755	2076	1384	2594
25	3050	2409	3816	4310	3930	1125	2513	2321	2750	1399
26	395	513	1155	1649	1269	1863	721	744	89	1262
27	697	316	1457	1951	1571	1705	739	504	391	1022
28	1328	600	1869	2363	1957	1202	291	991	1022	1148
29	2625	1984	3391	3885	3505	620	2088	1896	2325	974
30	1552	824	2093	2587	2181	1426	515	1215	1246	1372
	1	2	3	4	5	6	7	8	9	10

21	2187	814	1502	930	2651	300	577	1649	4002	523
22	2032	1253	1154	934	2850	1117	774	1848	4201	756
23	2387	1320	971	309	1919	500	777	917	3270	723
24	388	1573	2819	2247	4293	1874	1697	3291	5644	1719
25	4006	2939	1396	1928	300	2119	2396	702	1651	2342
26	1345	278	1487	915	2961	542	365	1959	4312	387
27	1647	580	1329	757	2721	302	180	1719	4072	202
28	2059	1211	826	606	2522	789	729	1520	3873	711
29	3581	2514	971	1503	933	1694	1971	277	2284	1917
30	2283	1435	1050	830	2746	1013	977	1744	4097	935
	11	12	13	14	15	6	17	18	19	20

21										
22	1417									
23	732	1250								
24	2174	2019	2374							
25	2351	2550	1619	3993						
26	842	975	1042	1332	2661					
27	602	954	802	1634	2421	302				
28	1089	328	928	2046	2222	933	913			
29	1926	2125	1194	3568	633	2236	1996	1797		
30	1313	358	1152	2270	2446	1157	1137	224	2021	
	21	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**

31	828	983	996	1490	771	2333	1094	1214	522	1732
32	490	540	1219	1713	1305	1894	626	817	184	1335
33	1681	948	2301	2795	2389	1422	729	1339	1375	1355
34	481	636	1032	1526	1118	1986	813	867	175	1385
35	606	302	1366	1860	1480	1691	552	579	300	1097
36	1397	826	1858	2352	1946	1428	325	1217	1091	1361
37	659	1254	862	1356	534	2604	1178	1346	793	2003
38	1244	511	1958	2452	2046	1113	380	902	938	1046
39	810	171	1570	2064	1658	1448	344	542	504	1029
40	881	572	1343	1837	1431	1728	235	943	575	1430
	1	**2**	**3**	4	5	**6**	**7**	**8**	**9**	**10**

31	1186	711	1957	1385	3431	1012	835	2429	4782	857
32	1440	373	1553	981	3034	615	392	2032	4385	414
33	2491	1564	1046	826	2745	1137	1101	1743	4096	1083
34	1222	364	1610	1038	3084	665	488	2082	4435	510
35	1556	489	1315	743	2796	377	154	1794	4147	176
36	2048	1280	1052	832	2748	1015	884	1746	4099	712
37	1052	776	2228	1656	3702	1283	1160	2700	5053	1128
38	2148	1127	737	517	2433	700	664	1431	3784	646
39	1760	693	1072	500	2768	340	215	1766	4119	197
40	1533	764	1352	901	3048	741	616	2046	4399	598
	11	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**

31	1312	1358	1512	1173	2731	470	772	1403	2706	1627
32	915	890	1115	1396	2734	95	397	960	2309	1184
33	1437	545	1135	2478	2442	1286	1283	438	2017	187
34	965	1077	1165	1209	2784	123	425	1056	2359	1280
35	677	767	877	1543	2496	211	192	722	2071	946
36	1309	102	1141	2035	2448	1002	1064	226	2023	256
37	1356	1442	1783	1039	3402	741	1043	1469	2977	1693
38	1000	417	826	2181	2133	849	844	89	1708	313
39	640	559	809	1747	2468	415	395	514	2043	738
40	1041	499	1210	1520	2748	486	796	526	2323	750
	21	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**

31										
32	534									
33	1756	1313								
34	347	187	1409							
35	681	238	1094	334						
36	1440	907	443	1093	877					
37	271	805	2027	618	952	1458				
38	1314	895	437	972	657	315	1590			
39	885	446	886	538	208	669	1156	449		
40	859	391	958	578	609	515	943	615	401	
	31	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**

41	1109	1670	885	1376	164	3020	1512	1796	1246	2419
42	1145	460	1911	2405	2025	1633	898	416	845	506
43	3237	2556	4003	4497	4117	1312	2700	2508	2937	1586
44	513	631	1065	1559	1151	1981	780	862	208	1380
45	461	497	1221	1715	1307	1808	704	678	155	1196
46	1407	538	2167	2661	2281	851	642	929	1101	784
47	967	1528	652	1146	254	2878	1370	1654	1067	2277
48	760	915	879	1373	967	2265	699	1123	454	1664
49	1126	1281	387	881	475	2631	1191	1512	820	2030
50	684	835	981	1475	1069	2185	651	1066	378	1584
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

41	1000	1226	2644	2072	4118	1699	1522	3116	5469	1544
42	2101	1034	1257	595	2205	214	491	1203	3556	437
43	4193	3126	1583	2115	113	2306	2583	889	1464	2529
44	1255	396	1605	1033	3079	660	483	2077	4430	505
45	1411	344	1432	938	2895	476	349	1893	4246	371
46	2357	1290	475	255	2171	727	719	1169	3522	665
47	770	1084	2502	1930	3976	1557	1380	2974	5327	1402
48	1069	643	1889	1317	3363	944	767	2361	4714	789
49	577	1009	2255	1683	3729	1310	1133	2727	5080	1155
50	1171	567	1809	1237	3283	864	687	2281	4634	709
	11	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**

41	1806	1776	2199	1062	3818	1157	1459	1803	3933	2027
42	446	1331	286	2088	1905	756	516	1003	1480	1227
43	2538	2737	1806	4180	187	2848	2608	2409	820	2633
44	960	1044	1160	1242	2779	118	420	1051	2354	1275
45	776	1041	976	1398	2595	66	236	898	2170	1122
46	1027	679	564	2344	1871	1012	854	351	1446	575
47	1664	1634	2054	829	3676	1015	1317	1661	3251	1885
48	1244	963	1444	1056	3063	402	704	990	2638	1214
49	1610	1455	1810	564	3429	768	1070	1482	3004	1706
50	1164	915	1364	1158	2983	322	624	942	2558	1166
	21	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**

41	687	1221	2235	1034	1368	1792	450	1892	1572	1277
42	1226	829	1351	879	591	1229	1497	914	554	955
43	3318	2875	2629	2971	2637	2635	3589	2320	2615	2935
44	380	154	1376	33	329	1060	651	967	533	545
45	536	161	1289	189	195	1029	809	852	360	552
46	1482	1043	571	1135	805	577	1753	262	597	877
47	545	1079	2187	892	1226	1650	308	1750	1430	1135
48	395	466	1422	279	613	979	479	1079	817	464
49	609	832	1914	645	979	1471	475	1571	1183	956
50	380	358	1374	203	533	931	581	1031	737	416
	31	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**

41										
42	1913									
43	4005	2092								
44	1067	874	2966							
45	1223	690	2782	184						
46	2169	850	2058	1130	957					
47	412	1771	3863	925	1081	2027				
48	813	1158	3250	306	468	1414	671			
49	636	1524	3616	678	834	1780	284	492		
50	915	1078	3170	204	388	1334	773	102	594	
	41	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**

51	1000	131	1760	2254	1874	1258	534	522	694	839
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

51	1950	883	882	310	2578	320	312	1576	3929	258
	11	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**

51	620	797	619	1937	2278	605	447	469	1853	693
	21	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**

51	1075	636	817	728	433	695	1346	380	190	591
	31	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**

51	1762	534	2425	723	550	407	1620	1007	1373	927
	41	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**

Apéndice C

Códigos Fuente

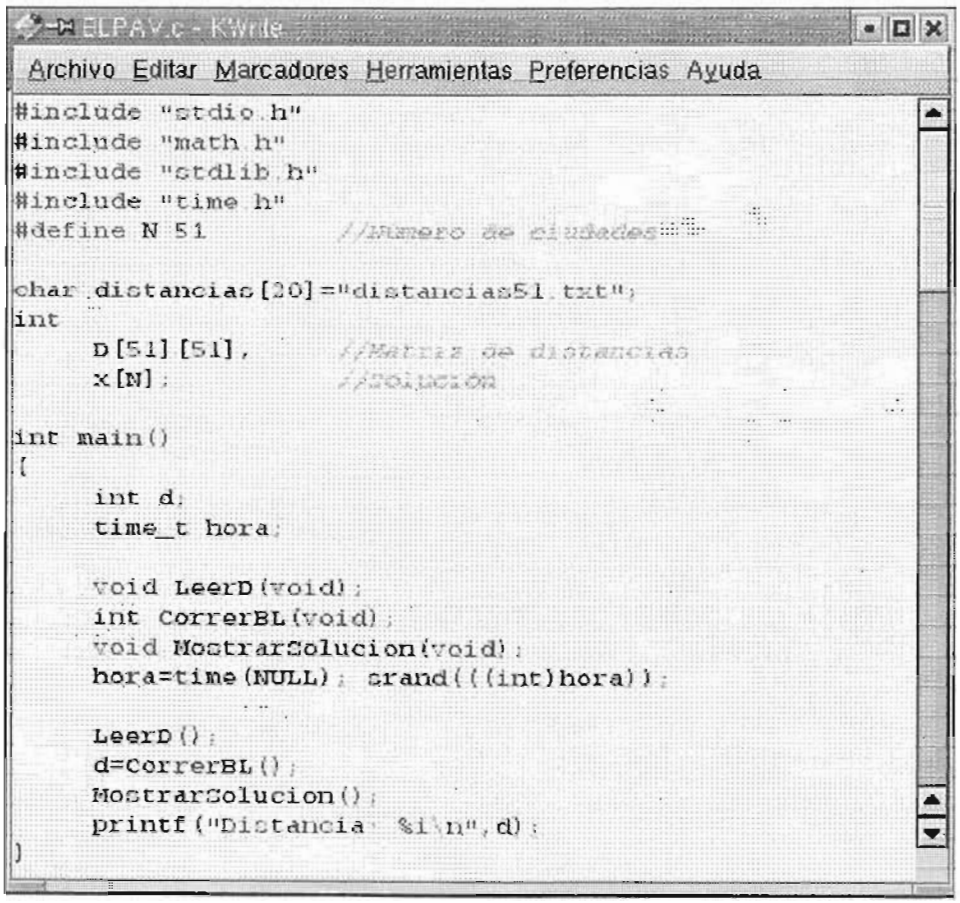
En este apéndice mostramos los códigos fuente, escritos en lenguaje C, para los algoritmos:

- Búsqueda local;.
- Búsqueda local modificada;
- Templado simulado;
- Templado simulado modificado;
- Máquina de Boltzmann secuencial;
- Máquina de Boltzmann secuencial modificada;

Los programas compilan bajo Linux. Para correr los programas en Windows posiblemente sea necesario hacer algunas pequeñas modificaciones en las cabeceras `define`.

C.1 BLPAV.c

El código fuente en C para el algoritmo búsqueda local es[1]:



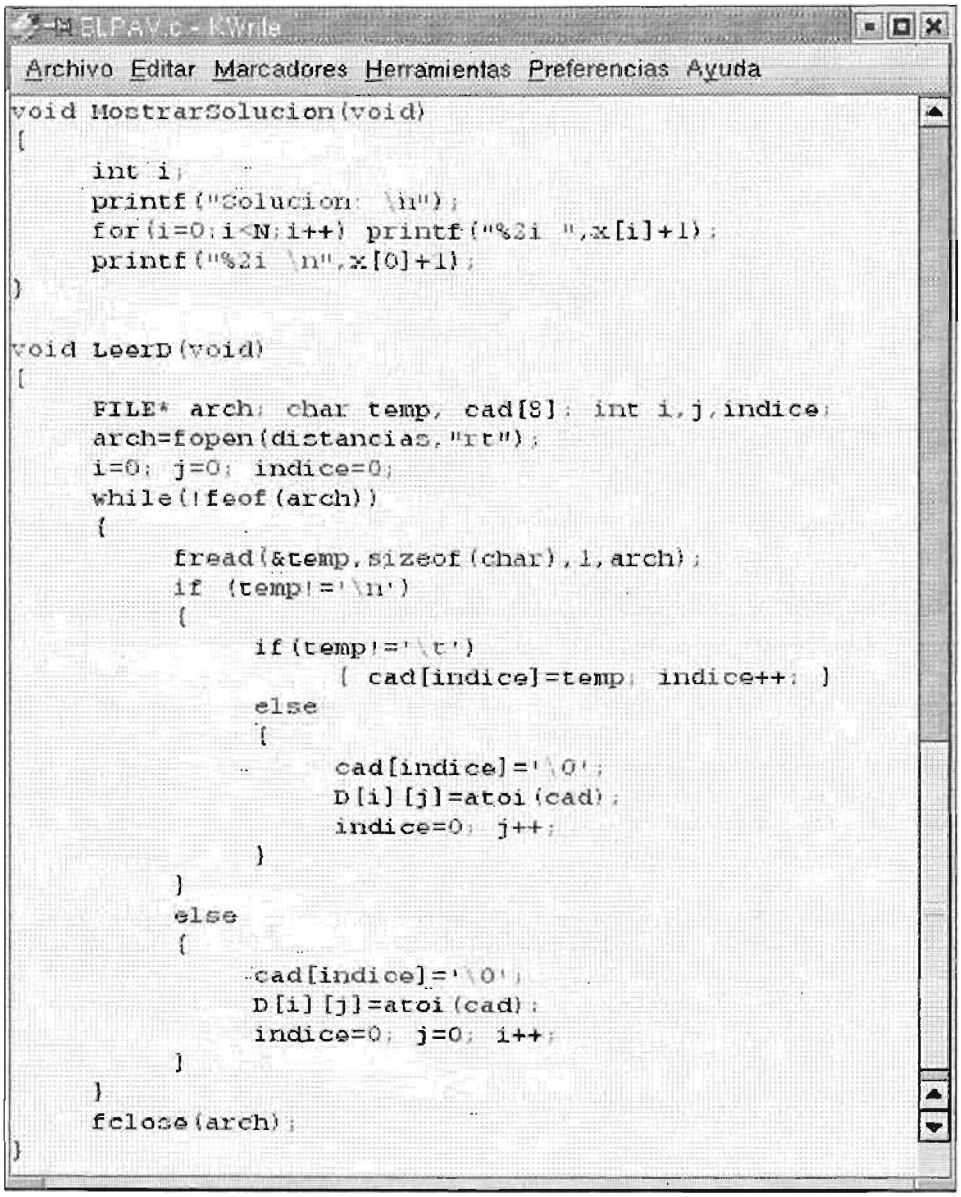
[1] Las demás funciones se encuentran en las siguientes páginas.

BLPAV.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
int CorrerBL(void)
{
     int dx,dy,xI[N],y[N],i,j,p,q,temp,pos;

     for(i=0;i<N;i++) xI[i]=-1;
     for (i=0;i<N;i++)
     {
          pos=rand()%N;
          while(xI[pos]!=-1) pos=rand()%N;
          xI[pos]=i;
     }
     for(i=0;i<N;i++) x[i]=xI[i];
     dx=0;
     for(i=0;i<N-1;i++) dx=dx+D[x[i]][x[i+1]];
     dx=dx+D[x[N-1]][x[0]];
     for(q=1;q<N-1;q++)
          for(p=q+1;p<N;p++)
          {
               for(i=0;i<N;i++) y[i]=xI[i];
               i=p; j=q;
               while(j<i)
               {
                    temp=y[i]; y[i]=y[j];y[j]=temp;
                    j++; i--;
               }
               dy=0;
               for(i=0;i<N-1;i++)dy=dy+D[y[i]][y[i+1]];
               dy=dy+D[y[N-1]][y[0]];
               if(dy<dx)
               {
                    for(i=0;i<N;i++) x[i]=y[i];
                    dx=dy;
               }
          }
     return dx;
}
```



ELPAV.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
void MostrarSolucion(void)
{
     int i;
     printf("Solucion: \n");
     for(i=0;i<N;i++) printf("%2i ",x[i]+1);
     printf("%2i \n",x[0]+1);
}

void LeerD(void)
{
     FILE* arch; char temp, cad[8]; int i,j,indice;
     arch=fopen(distancias,"rt");
     i=0; j=0; indice=0;
     while(!feof(arch))
     {
          fread(&temp,sizeof(char),1,arch);
          if (temp!='\n')
          {
               if(temp!='\t')
                    { cad[indice]=temp; indice++; }
               else
               {
                    cad[indice]='\0';
                    D[i][j]=atoi(cad);
                    indice=0; j++;
               }
          }
          else
          {
               cad[indice]='\0';
               D[i][j]=atoi(cad);
               indice=0; j=0; i++;
          }
     }
     fclose(arch);
}
```

C.2 BLMPAV.c

El código fuente en C para el algoritmo búsqueda local modificada es[2]:

BLMPAV.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
#include "stdio.h"
#include "math.h"
#include "stdlib.h"
#include "time.h"
#define N 51          //Numero de ciudades

const int
     K=10*N*N;        //Parametro de convergencia
char distancias[20]="distancias51.txt";
int
     D[51][51],       //Matriz de distancias
     x[N+1];          //Solución

int main()
{
     int d;
     time_t hora;

     void LeerD(void);
     int CorrerBLM(void);
     void MostrarSolucion(void);
     hora=time(NULL); srand(((int)hora));

     LeerD();
     d=CorrerBLM();
     MostrarSolucion();
     printf("Distancia: %i\n",d);
}
```

[2]Las funciones LeerD y MostrarSolucion aparecen en BLPAV.C La función CorrerBLM se encuentra en la siguiente página.

BLMPAV.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
int CorrerBLM(void)
{
     int d,d1,d2,dif,i,p,q,temp,pos,conv,cambio;

     for(i=0;i<N;i++) x[i]=-1;
     for (i=0;i<N;i++)
     {
          pos=rand()%N; while(x[pos]!=-1) pos=rand()%N;
          x[pos]=i;
     }
     x[N]=x[0];
     d=0; for(i=0;i<N;i++) d=d+D[x[i]][x[i+1]];
     conv=0;
     while(conv<K)
     {
          cambio=0;
          p=(rand()%(N-1))+1;
          q=p; while(p==q) q=(rand()%(N-1))+1;
          if(p<q){ temp=p; p=q; q=temp;}
          d1=D[x[q-1]][x[q]]+D[x[p]][x[p+1]];
          d2=D[x[q-1]][x[p]]+D[x[q]][x[p+1]];
          dif=d2-d1;
          if(dif<=0)
          {
               while(q<p)
               {
                    temp=x[p]; x[p]=x[q]; x[q]=temp;
                    q++; p--;
               }
               d=d+dif;
               if(dif<0) cambio=1;
          }
          if(cambio==0) conv++; else conv=0;
     }
     return d;
}
```

C.3 TSPAV.c

El código fuente en C para el algoritmo templado simulado es[3]:

TSPAV.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
#include "stdio.h"
#include "math.h"
#include "stdlib.h"
#include "time.h"
#define N 51          //Numero de ciudades

const int
     c0=5000000,      //Parametro de control inicial
     K=10*N;          //Parametro de convergencia
char distancias[20]="distancias51.txt";
int
     D[51][51],       //Matriz de distancias
     x[N+1];          //Solucion

int main()
{
     int d;
     time_t hora;

     void LeerD(void);
     int CorrerTS(void);
     void MostrarSolucion(void);
     hora=time(NULL); srand(((int)hora));

     LeerD();
     d=CorrerTS();
     MostrarSolucion();
     printf("Distancia: %i\n",d);
}
```

[3]Las funciones LeerD y MostrarSolucion se encuentran en BLPAV.c. La función CorrerTS se encuentra en la siguiente página.

TSPAV.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
int CorrerTS(void)
{
     int d,d1,d2,dif,i,cont,p,q,temp,pos,t,conv,cambio;
     double ct;
     for(i=0;i<N;i++) x[i]=-1;
     for (i=0;i<N;i++)
     {    pos=rand()%N; while(x[pos]!=-1) pos=rand()%N;
          x[pos]=i;
     }
     x[N]=x[0];
     d=0; for(i=0;i<N;i++) d=d+D[x[i]][x[i+1]];
     t=0; ct=c0; conv=0;
     while(conv<K)
     {    cambio=0;
          for(cont=0;cont<N;cont++)
          {    p=(rand()%(N-1))+1;
               q=p; while(p==q) q=(rand()%(N-1))+1;
               if(p<q) { temp=p; p=q; q=temp;}
               d1=D[x[q-1]][x[q]]+D[x[p]][x[p+1]];
               d2=D[x[q-1]][x[p]]+D[x[q]][x[p+1]]; dif=d2-d1;
               if(dif<=0)
               {    while(q < p)
                    {    temp=x[p]; x[p]=x[q]; x[q]=temp;
                         q++; p--;
                    }
                    d=d+dif;
                    if(dif<0) cambio=1;
               }
               else if(exp(-dif/ct)>((double)rand()/RAND_MAX))
               {    while(q < p)
                    {    temp=x[p]; x[p]=x[q]; x[q]=temp;
                         q++; p--;
                    }
                    d=d+dif;  cambio=1;
               }
          }
          if (cambio==0) conv++; else conv=0;
          t++; ct=0.95*ct;
     }
     return d;
}
```

C.4 TSMPAV.c

El código fuente en C para el algoritmo templado simulado modificado es[4]:

TSMPAV.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
#include "stdio.h"
#include "math.h"
#include "stdlib.h"
#include "time.h"
#define N 51          //Numero de ciudades

const int
     c0=5000000,      //Parametro de control inicial
     K=10*N;          //Parametro de convergencia
char distancias[20]="distancias51.txt";
int
     D[51][51],       //Matriz de distancias
     x[N+1];          //Solucion

int main()
{
     int d;
     time_t hora;

     void LeerD(void);
     int CorrerTSM(void);
     void MostrarSolucion(void);
     hora=time(NULL); srand(((int)hora));

     LeerD();
     d=CorrerTSM();
     MostrarSolucion();
     printf("Distancia: %i\n",d);
}
```

[4]Las funciones LeerD y MostrarSolucion se encuentran en BLPAV.c. La función CorrerTSM se encuentra en la siguiente página.

TSMFAV.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
int CorrerTSM(void)
{
    int d,dmin,d1,d2,dif,i,cont,p,q,temp,pos,t,conv,cambio,y[N+1];
    double ct;
    for(i=0;i<N;i++) y[i]=-1;
    for (i=0;i<N;i++)
    { pos=rand()%N; while(y[pos]!=-1) pos=rand()%N; y[pos]=i;}
    y[N]=y[0];
    d=0; for(i=0;i<N;i++) d=d+D[y[i]][y[i+1]];dmin=d;
    for(i=0;i<N+1;i++) x[i]=y[i];      //Guardar solucion
    t=0; ct=c0; conv=0;
    while(conv<K)
    {    cambio=0;
         for(cont=0;cont<N;cont++)
         {    p=(rand()%(N-1))+1;
              q=p; while(p==q) q=(rand()%(N-1))+1;
              if(p<q) { temp=p; p=q; q=temp;}
              d1=D[y[q-1]][y[q]]+D[y[p]][y[p+1]];
              d2=D[y[q-1]][y[p]]+D[y[q]][y[p+1]]; dif=d2-d1;
              if(dif<=0)
              {    while(q < p)
                   {    temp=y[p]; y[p]=y[q]; y[q]=temp;
                        q++; p--;
                   }
                   d=d+dif;
                   if(dif<0) cambio=1;
                   if (d<dmin)     //Guardar solucion
                   {
                        dmin=d; for(i=0;i<N+1;i++) x[i]=y[i];
                   }
              }
              else if(exp(-dif/ct)>((double)rand()/RAND_MAX))
              {    while(q < p)
                   {    temp=y[p]; y[p]=y[q]; y[q]=temp;
                        q++; p--;
                   }
                   d=d+dif; cambio=1;
              }
         }
         if (cambio==0) conv++; else conv=0;
         t++; ct=(double)c0/(1+t);
    }
    return dmin;
}
```

C.5 MBSPAV.c

El código fuente en C para el algoritmo máquina de Boltzmann secuencial es[5]:

MBSPAV.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
#include "stdio.h"
#include "math.h"
#include "stdlib.h"
#include "time.h"
#define N1 10          //Numero de ciudades

const int
     N=N1*N1,          //Numero de neuronas
     c0=10000000,      //Parametro de control inicial
     K=10;             //Parametro de convergencia
char distancias[20]="distancias51.txt";
int
     D[N1][N1],        //Matriz de distancias
     a[N],             //Patron de activacion
     W[N][N],          //Matriz de pesos
     x[N1][N1];        //Matriz solución

int main()
{
     int d;
     time_t hora;

     void LeerD(void);
     void CorrerMBS(void);
     void ConstruirSolucion(void);
     int CalcularDistancia(void);

     hora=time(NULL); srand(((int)hora));
     LeerD();
     CorrerMBS();
     ConstruirSolucion();
     d=CalcularDistancia();
     printf("Distancia: %i\n",d);
}
```

[5] La función LeerD se encuentra en BLPAV.c. Las demás funciones se encuentran en las siguientes páginas.

MBSP&V.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
void CorrerMBS()
{
     int conv,t,difC,cont,i,j,cambio,Net;
     double ct,F;
     void ConstruirW(void);

     ConstruirW();
     for(i=0;i<N;i++)
          a[i]=rand()%2;
     t=0; ct=c0; conv=0;
     while(conv<K)
     {
          cambio=0;
          for(cont=0;cont<N;cont++)
          {
               i=rand()%N;
               Net=0;
               for (j=0;j<N;j++)
                    Net=Net+W[i][j]*a[j];
               difC=(1-2*a[i])*(Net+(1-a[i])*W[i][i]);
               F=1/(1+exp(-(double)difC/ct));
               if (F>((double)rand()/RAND_MAX))
               {
                    a[i]=1-a[i];
                    cambio=1;
               }
          }
          if(cambio==0)
               conv++;
          else
               conv=0;
          t++;
          ct=(double)c0/(1+t);
     }
}
```

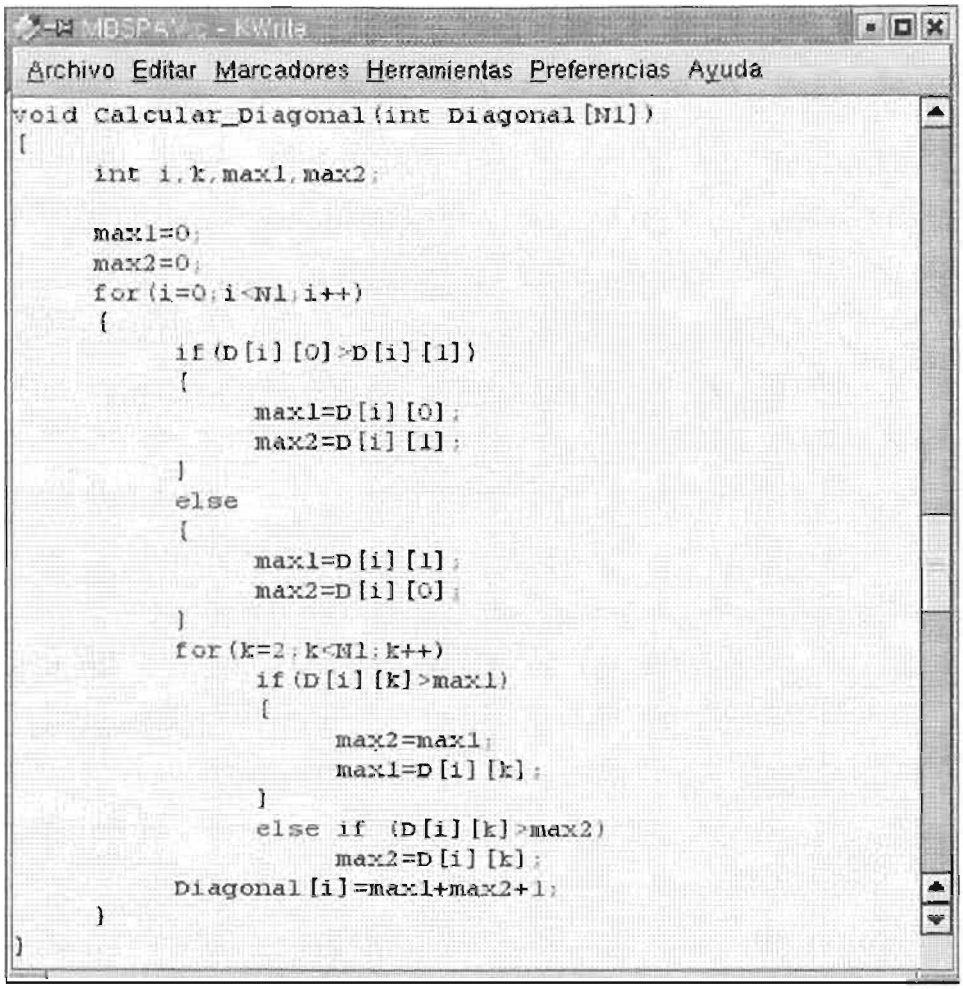
MBSPAV.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
void ConstruirW(void)
{
     int diagonal[Nl];
     int i,j,p,q,r,c,wij;
     void Calcular_Diagonal(int *);

     r=0;
     c=0;
     Calcular_Diagonal(diagonal);

     for(i=0;i<Nl;i++)
     for(p=0;p<Nl;p++)
     {
          for(j=0;j<Nl;j++)
          for(q=0;q<Nl;q++)
          {
               wij=0;
               if((i==j)&&(p==q))
                    wij=diagonal[i];
               else if((i!=j) && (q == (p+1) % Nl))
                    wij=-D[i][j];
               else if((j!=i) && (p == (q+1) % Nl))
                    wij=-D[i][j];
               else if (((i==j) && (p!=q))
                                   ||((i!=j) && (p==q)))
                    if (diagonal[i] < diagonal[j])
                         wij=-(diagonal[i]+1);
                    else
                         wij=-(diagonal[j]+1);
               W[r][c]=wij;
               c++;
          }
          r++;
          c=0;
     }
}
```




```
void Calcular_Diagonal(int Diagonal[N1])
{
     int i,k,max1,max2;

     max1=0;
     max2=0;
     for(i=0;i<N1;i++)
     {
          if(D[i][0]>D[i][1])
          {
               max1=D[i][0];
               max2=D[i][1];
          }
          else
          {
               max1=D[i][1];
               max2=D[i][0];
          }
          for(k=2;k<N1;k++)
               if(D[i][k]>max1)
               {
                    max2=max1;
                    max1=D[i][k];
               }
               else if (D[i][k]>max2)
                    max2=D[i][k];
          Diagonal[i]=max1+max2+1;
     }
}
```

MBSPAV.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
void construirSolucion()
{
     int i,j,indice;
     indice=0;

     printf("Solucion  \n");
     for(i=0;i<N1;i++)
     {
          for (j=0;j<N1;j++)
          {
               x[i][j]=a[indice];
               indice++;
               printf("%2i",x[i][j]);
          }
          printf("\n");
     }
}

int calcularDistancia()
{
     int i,j,p,q,d,aijpq;
     d=0;
     for(i=0;i<N1;i++)
     for (j=0;j<N1;j++)
          for(p=0;p<N1;p++)
          for (q=0;q<N1;q++)
               if (q==(p+1)%N1)
               {
                    aijpq=D[i][j];
                    d=d+(aijpq*x[i][p]*x[j][q]);
               }
      return d;
}
```

C.6 MBSMPAV.c

El código fuente en C para el algoritmo máquina de Boltzmann secuencial modificada es[6]:

MBSMPAV.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
#include "stdio.h"
#include "math.h"
#include "stdlib.h"
#include "time.h"
#define Ni 10          //Numero de ciudades

const int
     N=Ni*Ni,          //Numero de neuronas
     c0=10000000,      //Parametro de control inicial
     K=10,             //Parametro de convergencia
     Const=49625;      //Constante distancia/tiempo
char distancias[20]="distancias51.txt";
int
     D[Ni][Ni],        //Matriz de distancias
     a[N],             //Patron de activacion=solucion
     W[N][N];          //Matriz de pesos

int main()
{
     int d;
     time_t hora;

     void LeerD(void);
     int CorrerMBSM(void);
     void MostrarSolucion(void);

     hora=time(NULL); srand((int)hora));
     LeerD();
     d=CorrerMBSM();
     MostrarSolucion();
     printf("Distancia  %i\n",d);
}
```

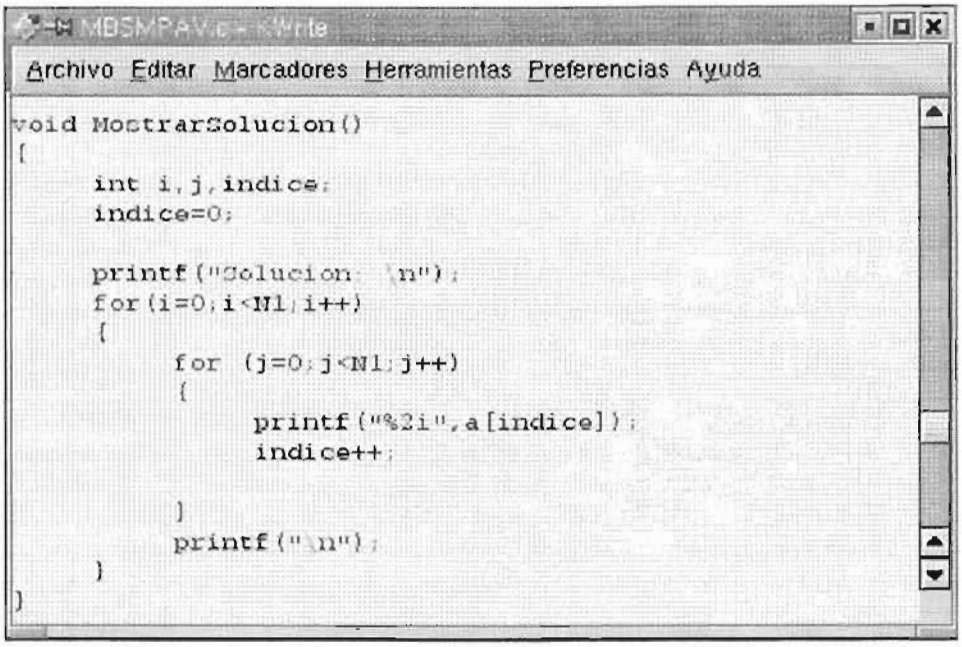
[6]La función LeerD se encuentra en BLPAV.c. Las funciones CorrerMBSM y MostrarSolucion se encuentran en las siguientes páginas. Las demás funciones se encuentran en MBSPAV.c.

MBSMPAV.c - KWrite

Archivo Editar Marcadores Herramientas Preferencias Ayuda

```
int correrMBSM()
{
     int cons,d,dmin,conv,t,difC,cont,i,j,cambio,Net,b[N];
     double ct,F; void construirW(void);

     construirW();
     for(i=0;i<N;i++) b[i]=rand()%2;
     cons=0;
     for(i=0;i<N;i++)
          for(j=i;j<N;j++)
               cons=cons+(W[i][j]*b[i]*b[j]);
     d=Const*cons;
     dmin=d; t=0; ct=c0; conv=0;
     while(conv<K)
     {
          cambio=0;
          for(cont=0;cont<N;cont++)
          {
               i=rand()%N;
               Net=0;
               for (j=0;j<N;j++)Net=Net+W[i][j]*b[j];
               difC=(1-2*b[i])*(Net+(1-b[i])*W[i][i]);
               F=1/(1+exp(-(double)difC/ct));
               if (F>((double)rand()/RAND_MAX))
               {
                    b[i]=1-b[i]; cambio=1;
                    cons=cons+difC;
                    d=Const*cons;
                    if(d<dmin) //Guardar solucion
                    {
                         dmin=d;
                         for(i=0;i<N;i++) a[i]=b[i];
                    }
               }
          }
          if(cambio==0) conv++;else conv=0;
          t++;
          ct=(double)c0/(1+t);
     }
     return dmin;
}
```


MBSMPAV.c - KWrite
Archivo Editar Marcadores Herramientas Preferencias Ayuda
void MostrarSolucion()
{
int i,j,indice;
indice=0;
printf("Solucion: \n");
for(i=0;i<N1;i++)
{
for (j=0;j<N1;j++)
{
printf("%2i",a[indice]);
indice++;
}
printf("\n");
}
}

Bibliografía

[1] AARTS EMILE; Simulated Annealing and Botlzmann Machines; Chinchester; Ed. John Wiley & sons;1990; 272 p.

[2] ACKLEY DAVID, HINTON GEOFFREY Y SEJNOWSKI TERRENCE; A learning algoruthm for Boltzmann machines , en Anderson James y Rosenfeld Edward, Neurocomputing fundations of research; Massachusetts institute; Ed. MIT; 1988; 729 p.

[3] ALDOUS JOAN M.; Graphs and applications : an introductory approach; London : Open University; Ed. Springer; 2000; 444 p.

[4] CERNY V.; Thermodynamical approach to the traveling salesman problem. Journal optimization theory and applications; New York; Ed. Plenum; 1985; 41-51p.

[5] CORMEN T., LEISERSON E. Y RIVEST R.; Introduction to Algorithms; McGraw-Hill; Singapore; 1990.

[6] CHARYRAND GARY Y OELLERMANND O.; Applied and Algorithmic Graph Theory; McGraw-Hill; Singapore; 1993; 395 p.

[7] CHRISTOFIDES NICOS; Graph theory : An algorithmic approach; London; Ed. Academic; 1975; 400 p.

[8] FOULDS LR.; Combinatorial optimization for undergraduates; USA; Ed. Springer-Verlag; 1948; 228 p.

[9] HECHT-NIELSEN ROBERT; Neurocomputing; Massachussets; Ed. Addison-Wesley; 1990; 433 p.

[10] HILERA JOSÉ; Redes Neuronales Artificiales; España; Ed. Alfaomega; 2000; 390 p.

[11] HOFFMAN P.; History. En Lawler E. et al , The Traveling salesman problem: A guided tour of combinatorial optimization; Chinchester; Ed. John Wiley; 1985; 413 p.

[12] HOPFIELD J.J.; Neural networks and physical systems with emergent collective computational abilities; USA; Ed. PNAS 79; 1982; 2554-2558 p.

[13] KATE SMITH; Neural Networks for combinatorial optimization: A review of More than a Decade of Research; Australia; Ed. Informs Journal on Computing Vol. 11, No. 1; 1999; 15-34 p.

[14] KOHONEN TEUVO; Self organization and associative memory; Berlin; Ed. Springer-Verlag; 1989; 312 p.

[15] KRIKPATRIK S.; Optimization by simulated Annealing; Sience Num. 220; 1983; 671-680 p.

[16] METROPOLIS N.; Equation of state calculations by fast computing machines. Journal of chemical physics 21; New York; Ed. American Institute of Physics; 1953; 1087-1092 p.

[17] PAPADIMITRIOU CRISTOS.; Combinatorial optimization: Algorithms and complexity; New Jersey; Ed. Prentice-Hall; 1982; 496 p.

[18] PARKER R. GARY; Discrete Optimization; Boston; Ed. Academic, 1988; 472 p.

[19] RAWLINS GREGORY; Compared to What? An Introduction to the Analysis of Algorithms; Computer Science Press; USA 1991; p.536.

[20] REEVES COLIN; Modern heuristic techniques for combinatorial problems; New York; Ed. Springer-Verlang; 1983, 320 p.

[21] RUMELHART D., HINTON G. Y MCCLELLAND J.; A General Framework for Parallel Distributed Processing. En Rumelhart D. Et al, Parallel distributed processing: Exoplorations in the Microstructure of congnition 1; Cambrige; Ed. MIT; 1986.

[22] TODA MORIKAZU; Statistical physics; Berlin; Ed. Springer; 1983.

[23] YASER S.; Neural Networks for computing?. En Denker J. Ed, Neural Networks for computing; New York; Ed. American Istitute of physics; 1986; 445 p.