Una implementación eficiente de algoritmo memético para TSP

Joel Artemio Morales Viscaya
Instituto Tecnológico de La Paz
División de estudios de posgrado e investigación
La Paz, Baja California Sur, México
iscviscaya@gmail.com

Marco Antonio Castro Liera
Instituto Tecnológico de La Paz
División de estudios de posgrado e investigación
La Paz, Baja California Sur, México
mcastro@itlp.edu.mx

Abstract **- Se presenta una implementación eficiente del algoritmo memético que se describe en una aplicación anterior para resolver el problema del agente viajero en clusters de computadoras. El algoritmo utiliza una estrategia de búsqueda local que mezcla características de 3-OPT y Lin-Kernighan. La implementación propuesta lleva a cabo la estrategia de búsqueda local directamente sobre las permutaciones que representan los tours sin necesidad de traducirlas a conjuntos de aristas. Los resultados experimentales muestran un rendimiento superior al de la implementación anterior al aplicarlo en diferentes instancias obtenidas de la TSPLIB de entre 150 y 1002 ciudades.**

Palabras clave- TSP(Traveling Salesman Problem), AG (Algoritmo Genético), arquitectura paralela, heurística, optimización, búsqueda local, computación evolutiva.

I. INTRODUCCIÓN

El problema del agente viajero simétrico (TSP por sus siglas en inglés, *Traveling Salesman Problem*) es un problema clásico y ampliamente estudiado de optimización combinatoria que consiste en encontrar, dado un conjunto de ciudades, la ruta más corta que permita a un agente viajero visitar todas ellas y regresar a la ciudad en la que inició el recorrido, en donde el costo del viaje entre las ciudades debe ser simétrica; es decir, el costo de ir de la ciudad A a la ciudad B debe ser el mismo que entre B y A, para cualquier par de ciudades en el conjunto [1].

TSP pertenece a la clase de problemas de optimización NP-duro (es un problema duro dentro de los no polinomiales) de acuerdo a [2] y es importante, pues constituye un componente central de un número considerable de problemas del mundo real, sus aplicaciones incluyen pero no están restringidas a la logística, la fabricación de circuitos integrados y tableros de circuitos perforados, medición por rayos x, data clustering, mira de telescopios, análisis de secuencias genéticas, guiado de láser para cristal, conexión de antenas, construcción, programación de tareas, la atención a llamadas de emergencia, servicio postal, entre otras.

TSP es posiblemente el problema más famoso y extensamente estudiado en el campo de la optimización combinatoria [3]. En años recientes, muchas heurísticas o meta-heurísticas han sido aplicadas para resolver problemas NP-duros (ya que por definición no existen algoritmos que los resuelvan en tiempos razonables), como por ejemplo el recocido simulado [4], la Búsqueda Tabú (TS) [5], Algoritmos Genéticos (AG) [6], Redes Neuronales (RN) [7], Optimización por Colonia de Hormigas (ACO) [8], Optimización por enjambre de partículas (PSO) [9], entre otras.

Un enfoque heurístico para los problemas de optimización combinatoria es la mejora iterativa de las soluciones, mediante una transformación que tome soluciones del problema y las lleve a nuevas soluciones con mejor aptitud, hasta que ya no sea posible encontrar dichas transformaciones, los algoritmos que llevan a cabo éstas transformaciones se conocen como de búsqueda local.

La combinación de algoritmos evolutivos poblacionales, cómo los mencionados anteriormente, con heurísticas de búsqueda local forman lo que se conoce como un algoritmo memético (MA), debido a la mezcla de las propiedades incluyentes de éstos 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 [2].

II. CLÚSTER DE COMPUTADORAS

Una alternativa para disminuir el tiempo de ejecución en la resolución de problemas complejos de ingeniería es el cómputo paralelo, que consiste en la ejecución de múltiples instrucciones de un programa simultáneamente en varias unidades de procesamiento [10], una forma de implementar el cómputo paralelo son los clúster de computadoras, que consisten en un conjunto de equipos de cómputo interconectados que colaborativamente intercambian y procesan información [11]. Sólo los procesadores locales tienen acceso directamente 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 [11], como se muestra en la Fig. 1

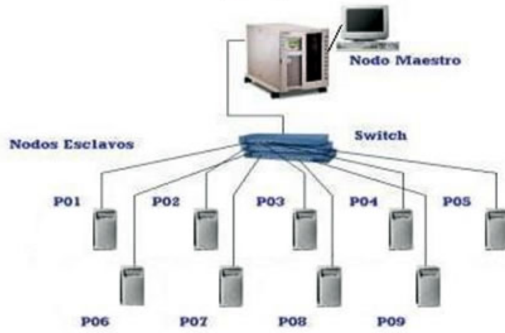

Figura 1. Esquema de un clúster de computadoras

III. EL ALGORITMO MEMÉTICO

En [1] se propone una estrategia en tres fases, una de inicialización de procesos en el clúster, la optimización que es un algoritmo memético y una etapa de consolidación de los resultados en que cada proceso envía hacia un proceso maestro, la fase fundamental o el núcleo de dicha estrategia es el algoritmo memético basado en un AG.

Un AG consta de tres operaciones básicas que se conocen como operadores genéticos: selección, cruza y mutación. Selección es el proceso por el cuál se elige que individuos o soluciones permaneceran en la generación siguiente, decidimos utilizar elitismo, que consiste en simplemente conservar a los mejores individuos de la población.

Cruza es como se le conoce al proceso que genera nuevas soluciones utilizando los individuos que superaron la etapa de selección.

La mutación se considera un operador secundario en los algoritmos genéticos. Se aplica solamente a una cantidad reducida de los individuos y produce cambios aleatorios grandes en sus elementos.

Además, cada cierto número de generaciones se envía una cantidad prefijada de individiduos (los mejores) hacia otra subpoblación, qué es lo que se conoce como migración.

Se modificó el ciclo principal de un AG, agregando después de los operadores genéticos de cruza y mutación, el algoritmo de búsqueda local, como se puede observar en la Fig. 2

III-A. Búsqueda local

Ya que se deben visitar todas las ciudades, TSP se puede reducir a encontrar el orden en que deben visitarse las mismas, lo cuál hace que la manera más barata de representar soluciones sea como un vector de n elementos, con n igual al número de ciudades de la instancia. Ésta representación en forma de permutación resulta particularmente adecuada para el AG, ya que existen múltiples algoritmos de cruza y mutación diseñados para permutaciones.

Sin embargo, aplicar los algoritmos de búsqueda local resulta más simple si representamos el problema en términos de grafos, vértices y aristas. El algoritmo de búsqueda local propuesto en [1] basado en hacer reemplazos de tres aristas, pero eligiendo qué aristas se van a reemplazar y, por qué aristas

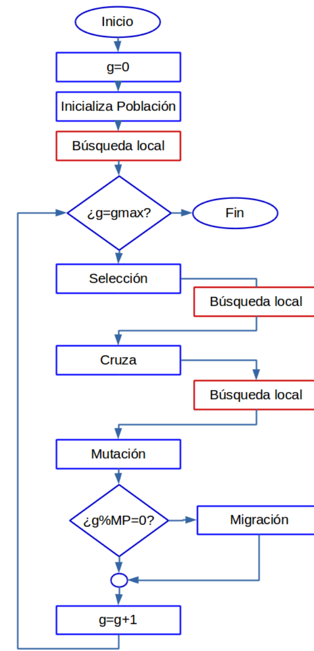

Figura 2. Diagrama del algoritmo memético

van a ser reemplazas de manera “inteligente”, de manera similar a cómo ocurre en LK, consta de los siguientes pasos:

1. Se transforma la permutación en un conjunto de aristas que forman la ruta o recorrido original.
2. Se elige al azar un nodo n_1.
3. 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.
4. 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.
5. 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.
6. Se elige de manera aleatoria $x_3 = (n_5, n_6)$, $x_3 \in T$, tal que $x_3 \neq x_1$, $x_3 \neq x_2$.
7. 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).
8. Si algún intercambio encontrado en 6 produce un costo menor al inicial, se efectúa el reemplazo y se regresa al paso 1.
9. En caso contrario, si ningun par y_2, y_3 produce un recorrido mejor, regresamos a 5 y seleccionamos otra

x_3.

10. 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 transformando el conjunto de aristas en una permutación.

En [1] se muestra que dicho algoritmo de búsqueda local produce mejores soluciones en tiempos más cortos que implementaciones eficientes de otras estrategias, sin embargo, el paso 7 resulta particularmente costoso ya que la verificación de que un recorrido sea Hamiltoniano es de orden $O(n^2)$.

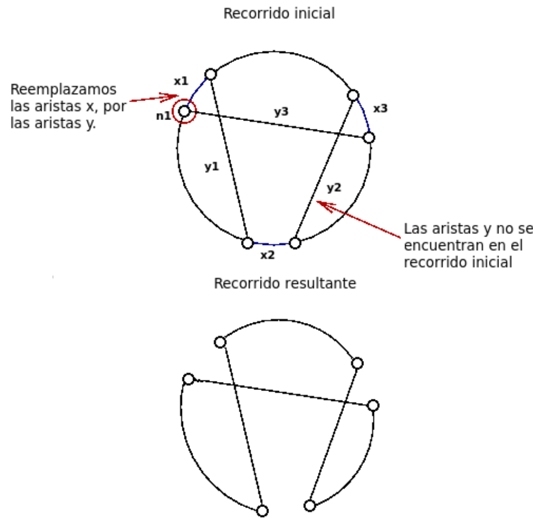

Figura 3. Ejemplo de iteración posible con reemplaza de tres aristas

Una posible iteración se puede observar en la Fig. 3

IV. Mejoras implementadas

Se modificó el algoritmo de búsqueda local por uno equivalente que trabaja directamente sobre la permutación. De esta forma no solamente nos ahorramos el primer y último paso, también eliminamos la costosa validación de que el recorrido resultante sea hamiltoniano. Esto debido a que toda permutación en que números adyacentes representen aristas, es un recorrido Hamiltoniano.

El algoritmo de búsqueda local mejorado consta de los siguientes pasos:

1. Se elige al azar una ciudad C_1 (una posición en la permutación).
2. Se selecciona una ciudad adyacente a C_1, C_2. La adyacencia entre $C_1 - C_2$ equivale a la arista x_1 en el algoritmo anterior.
3. Elegimos C_3 una ciudad tal que el costo de $C_2 - C_3$ sea menor al costo de $C1 - C2$ la conexión hipotética $C_2 - C_3$ equivale a la arista y_1 del algoritmo anterior, si no se puede encontrar C_3, regresamos a 2 y seleccionamos la otra ciudad adyacente a C_1, si no se puede encontrar C_3 tampoco en ese caso, regresamos al paso 1 y se selecciona otro nodo C_1.
4. Elegimos C_4 una ciudad tal que en la permutación original aparezca conectada a C_3. La conexión $C_3 - C_4$ es equivalente a x_2.
5. Se divide la permutación en tres segmentos, tales que algunas inversiones de dichos segmentos sean equivalentes a cambios de tres aristas, además las permutaciones resultantes de dichas inversiones no posean en índices adyacentes las ciudades C_3 y C_4 o las ciudades C_1 y C_2 equivalentes a x_1 y x_2, y en ella aparezcan en índices adyacentes las ciudades C_2 y C_3 (la arista y_1).
6. Si alguna de las inversiones del paso anterior produce un tour con costo menor que el original, se realiza la inversión, en caso contrario regresamos al paso anterior y probamos con otra división posible de la permutación (equivalente a seleccionar otro nodo x_3).
7. En caso de que ninguna partición del intervalo produzca inversiones con costo menor, regresamos al paso 4 y seleccionamos la otra ciudad C_4 posible.
8. En caso de haber probado con todas las ciudades C_4 posibles para un conjunto C_1, C_2, C_3 regresamos al paso 3 y elegimos otra ciudad C_3 posible.
9. En caso de haber probado con todas las ciudades C_3 posibles, regresamos al paso 2 e intentamos con la otra ciudad adyacente a C_1.
10. En caso de haber probado con todas las ciudades C_2 posibles, regresamos al paso 1 e intentamos con otra ciudad C_1, si ya probamos con todas las ciudades C_1 el algoritmo finaliza.

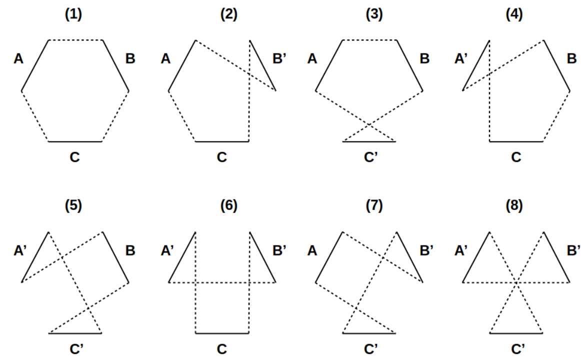

Figura 4. Posibles reconexiones con cambio de tres aristas

El paso 5 se puede llevar a cabo dado que, tal como se muestra en la figura 4, existen ocho formas en que puede reconectarse un grafo para producir un recorrido hamiltoniano al reemplazar 3 aristas [12]. Los segmentos a reconectar (marcados como A, B y C) en algunos casos deben recorrerse en sentido inverso (lo que se denota con una tilde) para formar la permutación que equivale al grafo resultante.

Otra mejora sobre la implementación anterior que nos permitió disminuir el tiempo de ejecución fue agregar un atributo booleano a los individuos que es utilizado para identificar aquellos que ya han pasado por el método de búsqueda local con el objetivo de evitar que individuos que ya se encuentran

en un óptimo local traten de volver a optimizarse localmente.

V. Implementación y pruebas

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.

Se eligieron nueve instancias de tamaño mediano (entre 150 y 1002 ciudades) de TSP para llevar a cabo las pruebas, dichas instancias se obtuvieron de la librería TSPLIB creada por el alemán G. Reinelt [13].

Las pruebas fueron llevadas a cabo en un clúster de 18 computadoras con procesadores i7-3770 con 8 GB de memoria RAM.

VI. Resultados experimentales

En la tabla I se presentan las nueve instancias utilizadas para probar la implementación del algoritmo memético con el costo de su recorrido óptimo.

Tabla I
Instancias de prueba

Instancia	Distancia óptima
kroA150	26,524
kroB150	26,130
kroA200	29,368
kroB200	29,437
pr226	80,369
pr264	49,135
pr299	48,191
pr439	107,217
pr1002	259,045

En la tabla II se muestran los parámetros con los que se llevaron a cabo los experimentos, donde, G es el número de generaciones que se ejecuta el algoritmo, TP el tamaño de cada población, NP el número de subpoblaciones o islas. En todos los casos se utilizó una migración de un sólo individuo en un período de tres generaciones, tamaño de elite uno, con una probabilidad de adaptación de 80 % y una probabilidad de mutación de 1 %.

Tabla II
Parámetros experimentales

Instancia	G	TP	NP
kroA150	25	15	32
kroB150	25	15	32
kroA200	20	20	64
kroB150	20	20	64
pr226	40	23	64
pr264	40	26	64
pr299	40	30	64
pr439	70	44	64
pr1002	300	30	144

Las tabla III y IV muestran los resultados obtenidos en [1] y con la implementación propuesta del algoritmo memético respectivamente, incluyendo en ésta última, la aceleración que produce respecto al tiempo de ejecución de la implementación anterior.

Tabla III
Resultados experimentales versión anterior

Instancia	ERP	Tiempo
kroA150	0.0 %	1.5s
kroB150	0.0 %	1.6s
kroA200	0.0 %	6.9s
kroB200	0.0 %	6.6s
pr226	0.0 %	14.8s
pr264	0.0 %	35.2s
pr299	0.0 %	30.8s
pr439	$5.9e^{-}6$ %	256.0s

Tabla IV
Resultados experimentales nueva versión

Instancia	ERP	Tiempo	Aceleración
kroA150	0.0 %	1.1s	36.6 %
kroB150	$1.8e^{-}4$ %	1.5s	6.6 %
kroA200	0.0 %	3.6s	91.6 %
kroB200	$7.5e^{-}5$ %	4.8s	37.5 %
pr226	0.0 %	3.3s	348.5 %
pr264	0.0 %	6.7s	425.3 %
pr299	0.0 %	20.4s	51 %
pr439	$5.6e^{-}4$ %	199.7s	28.2 %
pr1002	$8e^{-}4$ %	259.0s	N/A

VII. Conclusiones y trabajo futuro

Los resultados demuestran que esta nueva implementación es mucho más eficiente que la anterior en las instancias evaluadas, ya que los tiempos de ejecución se reducen en todas las pruebas, en algunos casos hasta cuatro veces, de acuerdo a las tablas III y IV .

Cabe resaltar que la instancia pr1002 resultó intratable bajo la implementación anterior, sin embargo esta nueva versión permitió resolverlo en un tiempo similar al que le toma resolver la instancia pr439 al algoritmo original propuesto en [1] y con un error relativo menor a $1e - 3\,\%$.

Actualmente se está trabajando en la implementación de este nuevo algoritmo en una arquitectura paralela distinta a los *clusters* como son las GPU, que recientemente han obtenido resultados prometedores con muchos algoritmos paralelos.

Otra cuestión que queda como trabajo futuro es la prueba de la implementación en instancias mayores de la TSPLIB.

VIII. Referencias

[1] J. Morales, M.A. Castro Liera, Un algoritmo memético paralelo para TSP, Instituto Tecnológico de La Paz, La Paz BCS, 2016.

[2] S. Arora, Polynomial time approximation schemes for Euclidean TSP and other geometric problems, J. ACM 45 (1998) 753−782.

[3] G. Gutin, A.P. Punnen, The Traveling Salesman Problem and its Variations, Kluwer Academic Publishers, Dordrecht, 2002.

[4] Y. Chen, P. Zhang, Optimized annealing of traveling salesman problem from the nth-nearest-neighbor distribution, Physica A 371 (2006) 627−632.

[5] J.Q. Li, Q.K. Pan, Y.C. Liang, An effective hybrid Tabu search algorithm for multi-objective flexible job shop scheduling problems, Comput. Ind. Eng. 59 (2010) 647−662.

[6] S.J. Louis, G. Li, Case injected genetic algorithms for traveling salesman problems, Inform. Sci. 122 (2000) 201−225.

[7] K.S. Leung, H.D. Jin, Z.B. Xu, An expanding self-organizing neural network for the travelling salesman problem, Neurocomputing 62 (2004) 267−292.

[8] M. Dorigo, L.M. Gambardella, Ant colonies for the traveling salesman problem, Biosystems 43 (1997) 73−81.

[9] X.H. Shi, Y.C. Liang, H.P. Lee, C. Lu, Q.X. Wang, Particle swarm optimization-based algorithms for TSP and generalized TSP, Inform. Process. Lett. 103 (2007) 169−176.

[10] Nvidia, Cuda C Programming Guide, Des. Guid., p 228, 2014.

[11] J. A. Castro, M. A. Castro Liera and I. Castro Liera, Programación paralela aplicada en optimización, 1st edition, La Paz, BCS 2012.

[12] A. Blazinkas, A. Misevicius, Combining 2-opt, 3-opt and 4-opt with k-swap-kick perturbations for the traveling salesman problem, Kaunas University of Technology, Department of Multimedia Engineering, 2011.

[13] G. Reinelt, TSPLIB, a traveling salesman problem library, ORSA J. Comput. 3 (1991) 376−384.