Traveling Salesman Problem - Ant Colony Optimization

Matías Irala[1] y Alejandro Elías[2]

***Abstract*— El problema del agente viajero (TSP) es uno de los más estudiados en el campo de la optimización, ya que cuenta con diversas aplicaciones en la industria. Hace referencia a la visita a lugares o nodos (para entregar o recoger mercancías) con el fin de resolver problemas que impidan minimizar algún objetivo (tiempo o costos) o maximizar algún otro. En el presente artículo se presenta una descripción general y una explicación de este problema a partir de la teoría de grafos y además, una implementación de la heurística de ACO (Ant Colony Optimization) paralelizada para hallar la solución con mayor rapidez y eficiencia.**

I. INTRODUCCIÓN

En el problema del agente viajero o problema del viajante, formalmente, dado un gráfico no dirigido con costos no negativos (simétricos) asociados a cada vértice, se requiere encontrar un ciclo hamiltoniano de costo mínimo [1]. A pesar de que una serie de variantes y generalizaciones de este problema han sido propuestos en la literatura, como el probabilístico TSP (pTSP) y el problema de enrutamiento del vehículo (VRP), en este documento nos enfocamos en resolver la forma estándar del TSP, tal como es comúnmente utilizado como un problema de referencia para evaluar el desempeño de metaheurística.

Los algoritmos de optimización de colonias de hormigas (ACO) fueron originalmente propuesto por Marco Dorigo en su tesis doctoral [2], inspirado en el comportamiento cooperativo de estos organismos vivos al buscar fuentes de alimentos. Las hormigas colocan un químico llamado feromona en las rutas para sugerir buenos caminos para que otros miembros de la colonia sigan, lo que con el tiempo converge en soluciones óptimas. El mapeo del algoritmo ACO para resolver el TSP es sencillo. Las hormigas se pueden pensar como agentes que se mueven de una ciudad a otra en el gráfico, eligiendo su próximo destino basado en una función probabilística. Inicialmente, m hormigas son colocados aleatoriamente en diferentes nodos del gráfico para completar sus rutas de forma independiente. Una vez que todas las hormigas terminan sus rutas, se selecciona la mejor (la que encontró el camino más corto) para colocarle feromona. Se realizan varias iteraciones de este procedimiento de tal manera que los mejores circuitos se fortalecen con químicos de cada iteración a otra.

Las metaheurísticas como ACO se propusieron originalmente como una solución para problemas combinatorios que son imposibles de resolver en tiempo polinómico [3]. En este sentido, las metaheurísticas son enfoques multi-objetivos, ya que apuntan a encontrar buenas soluciones (tan cerca de los óptimos locales) en un período de tiempo razonable, especialmente en aplicaciones del mundo real donde los tiempos de respuesta necesitan ser bajos.

En este artículo proponemos un algoritmo ACO paralelo (PACO) para resolver el TSP. Realizamos un comparativo estudio del rendimiento del algoritmo frente a su versión secuencial (SACO), en términos de consumo de CPU en varias instancias de referencia bien conocidas.

II. METODOLOGÍA

El método propuesto consiste en un esquema de paralelización para el algoritmo ACO original con el objetivo de acelerarlo preservando la consistencia de las soluciones encontradas.

II-A. Algoritmo de Optimización de Colonia de Hormigas

Como se indicó anteriormente, la aplicación de algoritmos ACO para resolver el TSP es sencilla. Sean $\tau_i j$ y $\eta_i j$, respectivamente, el nivel de feromona y la visibilidad en el vértice (i, j), los parámetros de entrada α y β que representan el

parámetro de influencia de feromona y el parámetro de influencia de visibilidad, respectivamente; N el número de iteraciones a realizar, y m la cantidad de hormigas a enrutar. El algoritmo ACO viene dado por el Algoritmo 1. De acuerdo con esta configuración, el algoritmo encamina cada hormiga secuencialmente, lo que en arquitecturas multinúcleo no aprovecha al máximo los recursos disponibles. Nuestro método tiene como objetivo abordar esta falla introduciendo un esquema de enrutamiento asíncrono.

Algorithm 1 ACO-TSP Algorithm

1: **for** each vertex (i, j) **do**
2: $\tau_{ij}(0) = \tau_0$
3: **end for**
4: **for** $k = 1 \rightarrow m$ **do**
5: Place ant k at random node.
6: **end for**
7: **for** $t = 1 \rightarrow N$ **do**
8: **for** $k = 1 \rightarrow m$ **do**
9: **repeat**
10: Choose next node according to
11: $P_{ij}^{k}(t) = \frac{[\tau_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{h \in J_i^k}[\tau_{ih}(t)]^{\alpha}[\eta_{ih}]^{\beta}}$
12: **until** Ant k completes its route
13: Store route length of ant k, L_k
14: **end for**
15: Store best route T^+ of length L^+
16: **for** each vertex (i, j) **do**
17: $\tau_{ij}(t) = (1 - \rho)\tau_{ij}(t) + \Delta\tau_{ij}(t) + e\Delta\tau_{ij}^{e}(t)$
18: **end for**
19: **end for**

Fig. 1: Algoritmo ACO-TSP

II-B. SACO

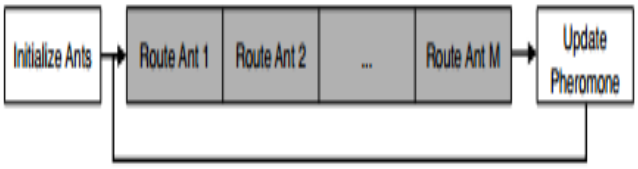

Fig. 2: Movimiento SACO

Como se especifica en el Algoritmo 1 y se muestra en la figura 1, se inicializan las hormigas ubicándolas en nodos aleatorios, luego el algoritmo encamina cada hormiga secuencialmente y al finalizar, se actualiza los niveles de feromonas.

En el siguiente fragmento de código se observa cómo se encamina cada hormiga secuencialmente. M es el número total de hormigas, cada valor de j representa una hormiga a ser encaminada, k representa cada paso que la hormiga da en su trayecto.

En el if, con las función hasCandidatesLeft analizamos los CLSIZE nodos más cercanos al nodo actual de la hormiga, si uno de ellos aún no ha sido visitado en su trayecto, entonces la hormiga escoge a uno de ellos aplicando la fórmula de probabilidad encontrada en la función argMax y el movimiento se realiza con la función moveAntTo.

Si los CLSIZE nodos más cercanos al nodo actual de la hormiga ya han sido visitados, no se aplica la fórmula de probabilidad y solo mueve la hormiga al nodo más cercano restante, esto en función de minimizar cálculos, como se observa en el else.

```
else {  //Modo secuencial
    for (j = 0; j < M; j++) {
        for (k = 1; k < ins.dimension; k++) {
            if (hasCandidatesLeft(j,ant[j].tour[k-1])) {
                moveAntTo(j, k, argMax(j, ant[j].tour[k-1]));
            }
            else {
                moveAntTo(j, k, NN(j,k-1));
            }
        }
    }
}
```

Fig. 3: ACO Secuencial

II-C. PACO

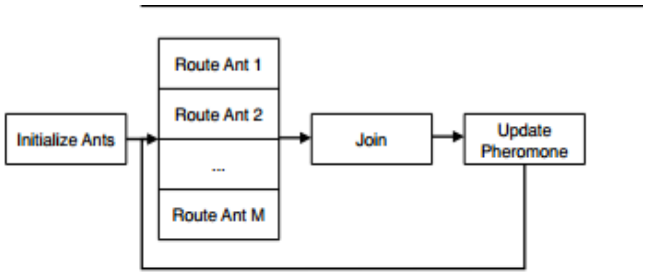

Fig. 4: Movimiento PACO

Para la paralelización, se aplican técnicas con la librería OpenMP, que consiste en ejecutar el procedimiento de enrutamiento de hormigas (líneas 9 a 12 en el Algoritmo 1) en diferentes hilos. Como

se muestra en el siguiente fragmento código, se utiliza OpenMP con la directiva #pragma omp for, que dividirá la cantidad total de hormigas en x cantidad de hilos (normalmente x sería la cantidad de núcleos del procesador). La paralelización simplemente divide el for con índice j en diferentes hilos independientes.

```
#pragma omp for
for (j = 0; j < M; j++) {   //Divide la cantidad total de h
    for (k = 1; k < ins.dimension; k++) {
        if (hasCandidatesLeft(j,ant[j].tour[k-1])) {
            moveAntTo(j, k, argMax(j, ant[j].tour[k-1]));
        }
        else {
            moveAntTo(j, k, NN(j,k-1));
        }
    }
}
```

Fig. 5: ACO Paralelo

III. Pasos para ejecutar el proyecto

1. Abrir una ventana de terminal en la carpeta principal del proyecto
2. Compilar el proyecto, para ello puede teclear en la terminal: make
3. Ejecute el proyecto con:
   ./tsp-acs <data> <number_of_ants> <use_threads>
   data es el directorio a una instancia TSP
   number_of_ants es el número de hormigas a ser enrutadas
   use_threas con valor 0 para secuencial y 1 para paralelo

Observaciones: El archivo con la instancia TSP posee una extensión .tsp, el cual corresponde al formato TSPLIB con tipo EUC_2D.

```
[dyuko@dyuko-hplaptop15db0xxx tsp-acs-master]$ make
gcc -lm -fopenmp -Wall -O3 src/tsp-acs.c -o tsp-acs
[dyuko@dyuko-hplaptop15db0xxx tsp-acs-master]$ ./tsp-acs data/a280.tsp 100 1
Running parallel version
Number of threads: 4
```

Fig. 6: Compilación y ejecución

IV. Análisis

Se ha realizado pruebas con las versiones secuencial y paralela del algoritmo ACO-TSP con el fin de evaluar su rendimiento en términos de precisión de la solución y tiempo de CPU. Tanto las versiones secuencial como paralela fueron codificadas en C; para la paralelización se ha utilizado la librería OpenMP.
Todas las simulaciones se han realizado en una máquina virtual Ubuntu, con procesador i5-8250U @1.60GHz Quad Core y 3 GB de memoria RAM. Para el análisis, se ha utilizado tres instancias TSP, donde se encuentran 76, 280 y 1002 ciudades (nodos) con sus puntos de coordenadas respectivos. Para la cantidad de hormigas a ser enrutadas, se han utilizado valores de 60, 120 y 240.
En las siguientes tablas se muestran los resultados obtenidos con las versiones secuencial y paralela, todos los tiempos se miden en segundos:

Ants = 60	SACO		PACO		SpeedUp	Solucion Real
Ciudad	Tiempo	Solucion	Tiempo	Solucion		
g76	3,424	545,97	1,409	544,37	2,43	538
g280	16,213	2605,016	6,401	2600,88	2,53	2579
g1002	111,313	331947,9	57,186	334550,687	1,94	259045

Fig. 7: Análisis con 60 hormigas

En el análisis con 60 hormigas se observa una gran mejoría de tiempo de ejecución utilizando PACO, casi en todos los casos con un SpeedUp mayor a 2.

Ants = 120	SACO		PACO		SpeedUp	Solucion Real
Ciudad	Tiempo	Solucion	Tiempo	Solucion		
g76	6,42	547,244	2,635	544,37	2,43	538
g280	30,3	2610,341	10,84	2595,344	2,79	2579
g1002	197,741	333949,531	88,398	334576,718	2,23	259045

Fig. 8: Análisis con 120 hormigas

Para el análisis con 120 hormigas, se observan que las soluciones no fluctúan notoriamente con respecto a la tabla anterior, lo que lleva a pensar en la existencia de un límite de hormigas por cada grafo, con el cual nos devuelva el valor óptimo posible para el mismo.
En este caso, ya que con 120 hormigas nos retorna soluciones un poco más alejadas que con 60 hormigas, se estima que los resultados mejoren aumentando el número de hormigas a 90 0 100.

Ants = 240	SACO		PACO		SpeedUp	Solucion Real
Ciudad	Tiempo	Solucion	Tiempo	Solucion		
g76	12,025	546,502	5,257	548,845	2,28	538
g280	58,421	2612,804	20,1	2607,67	2,9	2579
g1002	395,525	329503,375	154,842	330769,406	2,55	259045

Fig. 9: Análisis con 240 hormigas

En la última tabla de análisis, correspondiente a 240 hormigas, se observa resultados semejantes a la tabla anterior. Los tiempos de ejecución mejoran mucho con respecto a la versión SACO, pero los resultados son mejores o peores dependiendo del caso.
El caso de g1002 de esta tabla, con 240 hormigas, es el resultado más óptimo comparando las tres tablas.

V. RESULTADOS

Se observa gran diferencia entre los tiempos de SACO y PACO en cada tabla, dando a entender que el método paralelo implementado está funcionando como debería hacerlo, haciendo que la ejecución del programa sea casi el doble de rápido en todos los casos.
Con respecto a los resultados obtenidos surge una interrogante, ¿por qué en algunos casos tenemos resultados más lejanos al resultado óptimo y en otros casos más cercanos? Un problema que viene con este algoritmo es el estancamiento [4]. Cuando muchas hormigas viajan en el espacio problemático, van a soltar muchas feromonas, pero como muchas de ellos gravitan hacia la mejor solución que encuentran, crea hambre en otras partes del conjunto de datos. Esto significa que en algún punto, todas las hormigas terminarán tomando aproximadamente el mismo camino, ya que la probabilidad de desviarse de ese camino se vuelve insignificante, casi nula. Esto es debido al hecho de que las feromonas se degradan con el tiempo y solo los caminos recorridos se reponen, lo que lleva a situaciones en las que no se pueden encontrar mejores soluciones, incluso si hay una o mas. Este es el problema de trabajar con este algoritmo. Es un excelente algoritmo para resolver problemas de recorrido, no hay duda, pero sus resultados no devuelven el valor óptimo, solo una aproximación.

VI. CONCLUSIÓN

En este artículo se ha propuesto un esquema de paralelización para el algoritmo ACO y se ha evaluado su desempeño usando instancias de benchmark del TSP en términos de consumo de tiempo de CPU y precisión de la solución. Se ha confirmado la consistencia relativa entre las versiones secuencial y paralela del algoritmo, en otras palabras, se ha verificado que sus resultados no difieren en una cantidad significativa, y las pequeñas diferencias son debido a la inicialización aleatoria de parámetros, que no pueden ser controladas [6]. Se ha observado que PACO supera significativamente el rendimiento de SACO en tiempo de CPU. Se concluye que PACO supera a SACO en tiempo de CPU en una solución de calidad comparable. Además se ha observado, que la cantidad de hormigas, en algunos casos, no influyen en el resultado esperado, como explicamos en la sección anterior. En este caso, se debería encontrar el número óptimo de hormigas para obtener los mejores resultados posibles.

REFERENCES

[1] El problema del viajante. Métodos de resolución y un enfoque hacia la teoría de computación, Salvador Peñalva García, 2015

[2] M. Dorigo. Optimization, Learning and Natural Algorithms. PhD thesis, Politecnico di Milano, Italy, 1992.

[3] Max Manfrin, Mauro Birattari, Thomas Sttzle, and Marco Dorigo. Parallel ant colony optimization for the traveling salesman problem. In Marco Dorigo, Luca Gambardella, Mauro Birattari, Alcherio Martinoli, Riccardo Poli, and Thomas Sttzle, editors, Ant Colony Optimization and Swarm Intelligence, volume 4150 of Lecture Notes in Computer Science, pages 224–234. Springer Berlin / Heidelberg, 2006. 10.1007/11839088 20.

[4] Ant Colony Optimization - Optimal Number of Ants, CHRISTOFFER LUNDELL JOHANSSON, LARS PETTERSSON, 2018

[5] Martn Pedemonte, Sergio Nesmachnow, and Hctor Cancela. A survey on parallel ant colony optimization. Applied Soft Computing, 11(8):5181 – 5197, 2011.

[6] Felipe Ramírez González. A Parallel Ant Colony Optimization Algorithm for the Traveling Salesman Problem. Universidad Técnica Federico Santa María. Chile. 2011