TSP paralelo

Alex Loja
Ciencia de la computación
Universidad de Tecnología e Ingeniería
alex.loja@utec.edu.pe

Jean Ángeles
Ciencia de la computación
Universidad de Tecnología e Ingeniería
jean.angeles@utec.edu.pe

Keywords–TSP, paralelo, branch and bound.

1. Introducción

El Problema del Viajante (TSP) es un desafío que consiste en encontrar el camino más corto y eficiente para recorrer todos los nodos de una lista. En este problema, cada par de nodos tiene asignado un peso que representa la distancia entre ellos.

Dado un conjunto de N ciudades y las distancias entre cada una de ellas, el objetivo es encontrar el camino más corto posible que visite cada ciudad una sola vez y regrese al punto de inicio. Este problema pertenece a la categoría de los problemas NP-Hard, lo que significa que no existe una solución de tiempo polinomial conocida. Una solución de fuerza bruta para este problema considera una ciudad o nodo 1 como punto de inicio, genera todas las permutaciones de las N - 1 ciudades restantes y devuelve la permutación con el menor costo. Sin embargo, la complejidad de esta solución es O(N!).

Existen otras soluciones, como la que utiliza programación dinámica, que tienen una complejidad de O($N^2 2^N$), pero presentan un crecimiento exponencial del espacio requerido.

En este experimento, nuestro objetivo es utilizar el algoritmo de Branch and Bound aplicado al TSP y analizar su rendimiento para distintos números de ciudades.

A. Implementación

Para abordar el problema utilizando el algoritmo de Branch and Bound, comenzamos utilizando la matriz de adyacencia del grafo y seleccionamos la ciudad 1 como punto de partida. En primer lugar, es necesario normalizar la matriz de adyacencia o costos, lo que implica asegurarnos de que cada fila y columna contengan al menos un valor de 0. Para lograr esto, reducimos el costo mínimo de cada fila y columna, y el costo total será la suma de todos los valores mínimos que se han sustraído. Una vez completada la normalización, seleccionamos el nodo con el menor costo y reducimos la matriz de adyacencia en consecuencia.

A continuación se muestra el algoritmo que utilizaremos en nuestra implementación. Este algoritmo hace uso de una cola de prioridad, que mantiene en primer lugar la ciudad con el menor costo hasta ese momento. Esto nos permitirá visitar solo el camino menos costoso en cada nivel.

Algorithm 1: TSPbranchandbound

Input: matriz de costos
Output: costo mínimo
Data: xqf
$i \leftarrow 10$
$ciudades \leftarrow vector$
$root \leftarrow primeraciudad$
$pq \rightarrow push(root)$
$n \leftarrow numerodeciudades$
while *pq not empty* **do**
  $min \leftarrow pq.top()$
  $pq.pop()$
  $i \leftarrow (min \rightarrow vertex)$
  if $(min \rightarrow level) == n - 1$ **then**
    return $min \rightarrow cost$
  while $j < n$ **do**
    if $matrix[i][j]! = INF$ **then**
      $child \leftarrow newcity$
      $child.path \rightarrow addj$
      $cnodes \leftarrow setInfDist(child.matrix, i, j)$
      $child.cost \leftarrow min.cost +$
       $normalizar(child.matriz) + cnodes$

B. Análisis

El análisis general del algoritmo propuesto se puede desglosar en las siguientes partes:

- Cálculo de la matriz reducida: El cálculo de la matriz reducida se realiza al comienzo del algoritmo y requiere recorrer cada fila y columna de la matriz original. Esto tiene una complejidad de O(N^2), donde N es el número de ciudades.
- Bucle principal: El bucle principal del algoritmo se ejecuta mientras la cola de prioridad no esté vacía. En cada iteración, se extrae el nodo con el menor costo de la cola de prioridad y se generan sus nodos hijos.
- Generación de nodos hijos: En cada iteración del bucle principal, se generan N nodos hijos correspondientes a las N posibles extensiones del recorrido actual. Esto tiene una complejidad de O(N).
- Cálculo de la cota superior: Para cada nodo hijo generado, se calcula su costo utilizando la función de cota superior. Esta operación tiene una complejidad de O(N) debido a

la necesidad de recorrer la matriz reducida.
- Inserción y extracción de la cola de prioridad: La inserción y extracción de elementos en la cola de prioridad tienen una complejidad de O(log n), donde n es el número de elementos en la cola. En cada iteración, se inserta un nodo hijo en la cola de prioridad y se extrae el nodo de menor costo.

La complejidad depende del número de ciudades y puede verse disminuida por las estrategias de poda. Sin embargo, en el peor de los casos se sigue esperando que sea O($n^2 * 2^n$).

2. Paralelización

Al analizar el algoritmo evaluamos las partes que podrían verse beneficiadas por el uso de la paralelización. En nuestro caso decidimos paralelizar la generación de los nodos hijos. En esta parte del código se crean todos los nodos hijos lo que implica calcular el costo de traslado entre todas las posibles ciudades a las que se puede ir.

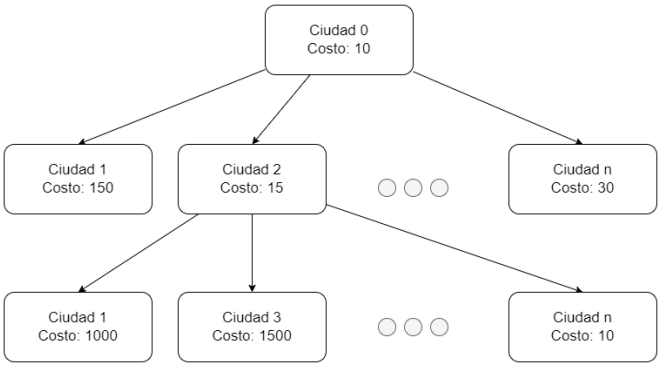

Fig 2.1: Árbol de ejecución

Como se puede obesrvar en **4.4**, por cada nivel se crean n - x nodos hijos, donde x es el nivel actual del nodo. Como estamos trabajando bajo un paradigma de memoria compartida tenemos que crear una zona crítica en nuestra región paralela, está será al momento en el que se necesite insertar un nuevo nodo hijo a la cola de prioridad.

```
#pragma omp parallel for shared(min) schedule(dynamic)
for (int j = 0; j < N; j++)
{
    if (min->matrix_reduced[i][j] != INF)
    {
        Node<T>* child = newNode ...;
        child->cost = ...;
        #pragma omp critical
        pq.push(child);
    }
}
}
```

Como no se harán cambios en el nodo mínimo, este puede ser compartido entre todos los procesos. Se probo paralelizar algunas partes además de está pero los resultados en tiempo de ejecución no mejoraron.

3. Especificaciones del sistema para el experimento

Table 3.1: Especificaciones de Hardware

Arquitectura	x86_64
Núcleos	4
Procesadores lógico	8
Socket(s)	1
Modelo	Intel(R) Core(TM) i7-6700HQ
Velocidad base	2.6 GHz

Table 3.2: Especificaciones de Software

Sistema operativo	Ubuntu 22.04.2 LTS
g++ Versión	11.3.0
OpenMP	5.1

*El sistema operativo es una virtualización ejecutada sobre Windows 10.

4. OpenMP - Procesamiento con memoria compartida

A. Comprobación de resultados

Para la experimentación se usaron en general matrices de distancias que implicaban simetria, es decir que la distancia entre un nodo y otro es la misma en ambas direcciones.[1]

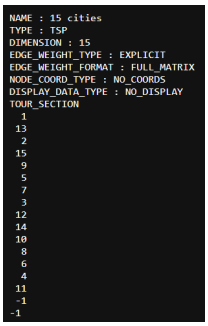

Fig 4.1: Camino esperado

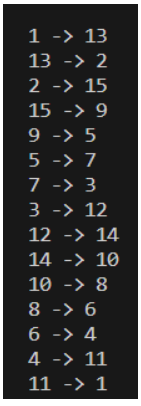

Fig 4.2: Camino obtenido

Para comprobar el correcto funcionamiento de nuestro algoritmo corrimos un dataset de 15 ciudades obteniendo el resultado esperado.

B. Análisis de tiempo

Las siguientes imágenes ilustran el tiempo que toma en cada ejecución y el speedup lograda por la paralelización usando OpenMP con diferente cantidad de hilos y números de ciudades.

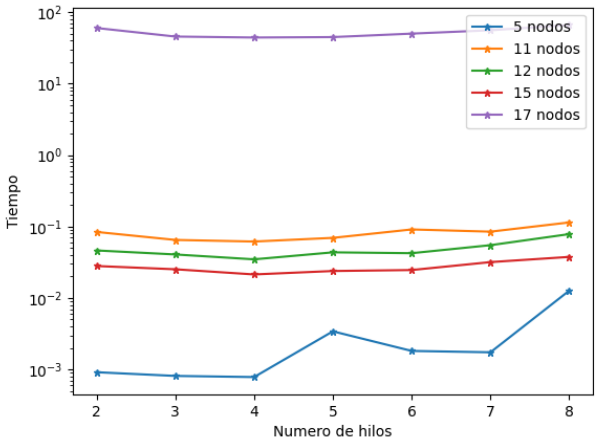

Fig 4.3: Variación del tiempo de ejecución con OpenMP para diferentes números de ciudades(Nodos)

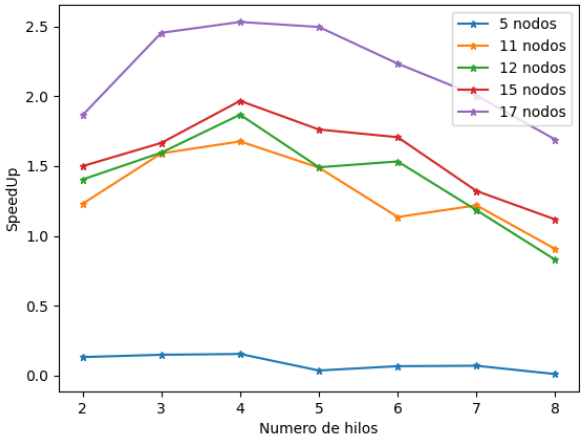

Fig 4.4: Variación del speedup logrado con OpenMP para diferentes números de ciudades(Nodos)

5. CONCLUSIONES

- El tiempo de ejecución decrece a medida que aumentamos el número de threads. Sin embargo a partir de 5 se observa como pierde eficiencia.
- La diferencia entre los tiempos de ejecución para diferentes números de ciudades es grande ya que depende de N!.
- El speedup aumenta a medida que crece el número de procesadores, en nuestro caso alcanzando su punto más alto para 4 hilos.
- El decrecimiento de performance a partir de 5 procesadores puede ser ocasionado por el número de núcleos con el que cuenta nuestro procesador que es 4.
- Para una cantidad pequeña de nodos se observa que no existe una mejoría notoria después del uso de paralelización.

REFERENCES

[1] G. Reinelt. Data for the traveling salesperson problem. Technical report, 2019.

Repositorio: https://github.com/aljozu/TSP