República Bolivariana de Venezuela
Ministerio del Poder Popular Para la Educación
Universidad Simón Bolívar

Reporte de laboratorio
Proyecto 1

Asignatura: CI-2692

Profesor:
Guillermo Palma

Hecho por:
Luis Isea. 19-10175
Juan Cuevas. 19-10056

Sartenejas, junio de 2023

Diseño de la solución

Para el diseño de nuestro proyecto usamos principalmente como tipo de datos a los arreglos de pares de números reales (Que llamaremos ruta), y arreglos de pares de pares de números reales (Que llamaremos ciclos). En un principio, nuestro código en DCLSTSP.kt toma como primer argumento (***args[0]***) el archivo de la librería TSP a resolver, y como segundo argumento (***args[1])*** el archivo donde se enviará la solución. Primeramente, lee el archivo ***args[0]*** con un lector almacenado en el búfer, en donde inicialmente lee las primeras 6 líneas, sin embargo, solo extrae la siguiente información de las líneas 1 y 4:

- Línea 1: Nombre de la instancia a resolver
- Línea 4: Dimensión (Diremos ***D***) de la instancia a resolver

Ya con la dimensión de nuestra instancia, creamos una ruta de tamaño ***D***, donde irán cada una de las ciudades, y cuyas coordenadas serán números reales representados con Double. Para esto, iteramos en las siguientes D líneas del archivo de la instancia y observamos el orden de cada línea:

Índice (coordenada en X) (coordenada en Y)

Ya que por los momentos no nos interesa el índice de cada ciudad, tomamos únicamente sus coordenadas y las agregamos al par, y dicho para a la ruta en su posición ***i***. Una vez tengamos nuestra ruta, creamos una copia de la misma que se mantendrá fija durante todo el proceso.

Con nuestra ruta ya creada, ejecutamos nuestra función ***divideAndConquerTSP*** para buscar el ciclo más óptimo posible con las ciudades que tenemos. Esta función se divide en 5 posibles casos:

- Si hay 0 ciudades en la ruta, entonces devuelve un ciclo vacío
- Si hay 1 ciudad en la ruta, entonces devuelve un ciclo que cubre esa ciudad
- Si hay 2 ciudades, entonces devuelve un ciclo que empieza en la primera ciudad, pasa por la segunda ciudad y regresa a la primera
- Si hay 3 ciudades, entonces devuelve un ciclo que empieza en la primera ciudad, pasa por la segunda ciudad, luego por la tercera ciudad y finalmente vuelve a la primera ciudad.
- Si hay más de 3 ciudades, entonces particionamos nuestra ruta en dos rutas más pequeñas, para luego crear un ciclo con cada ruta y unir ambos resultados.

Los primeros 4 casos son triviales ya que no ejecutan otras acciones. Caso contrario a cuando tenemos más de 3 ciudades.
En este caso particionamos en 2 rutas nuestra ruta principal, agrupando en base a el eje X o Y a las coordenadas de cada ciudad, además, tomamos un punto medio que nos sirva para dividir ambas rutas.
Ya con esto tendríamos que ir resolviendo recursivamente cada ruta y convertirla en un ciclo viable, para después unirlo con los demás ciclos de la manera más óptima.
Si bien suena fácil, calcular este ciclo es muy complicado. Aun con el tiempo que le hemos dedicado, aún no nos hemos podido acercar del todo al resultado más óptimo para cada prueba.

Resultados Experimentales

Se probó en un computador portátil con las siguientes características: Intel® Celeron® CPU N2830 @ 2.16GHz de doble núcleo (4gb de RAM), Kotlin versión 1.8.20-release-327 (JRE 17.0.6+10), OpenJDK 64-Bit Server VM Temurin-17.0.6+10.

Nombre de la instancia	Distancia Obtenida	% Dev. Valor óptimo
berlin52	23.356	209,68%
ch130	21.171	246,50%
ch150	23.182	255,12%
d1291	208.494	310,41%
d1655	234.922	278.13%
d198	57.028	261.39%
d2103	299.262	271.99%
d493	131.600	275.98%
d657	191.891	292.32%
eil101	1.745	177.42%
eil51	1.119	162.68%
eil76	1.400	160.22%
fl1400	133.627	563.92%

fl1577	139.325	526.21%
fl417	69.693	487.58%
gil262	8.843	271,86%
kroA100	72.940	242.73%
kroA150	95.617	321.98%
kroA200	111.927	281,12%
kroB100	73.409	231,55%
kroB150	98.799	278.11%
kroC100	76.190	267.20%
kroD100	76.498	259.24%
kroE100	75.448	241.88%
lin105	52.043	261.94%
lin318	170.423	305.49
p654	208.035	500.51%
pcb1173	237.647	317.72%
pcb3038	568.776	313.07%
pcb442	172.078	238.88%
pr1002	1.092.676	321.81%
pr107	122.395	176.27%
pr136	274.253	183.39%
pr144	265.593	353.72%
pr152	260.050	252.94%
pr226	318.939	296.84%
pr2392	1.640.017	333.83%
pr264	147.605	200.40%
pr299	189.134	292.46%
pr439	511.486	377.05%

pr76	310.644	187.20%
rat575	24.546	262.40%
rat783	33.363	278.86%
rd100	26.160	230.72%
rd400	58.693	284.09%
rl1304	1.293.011	411.17%
rl1323	1.328.802	391.78%
rl1889	1.649.922	421.24%
rl5934	2.764.828	397.23%
st70	1.924	185.03%
tsp225	13.329	240.37%
u1060	981.341	337.91%
u159	155.179	268.77%
u1817	224.146	291.85%
u2319	633.832	170.57%
u574	148.204	301.58%
u724	166.700	297.75%
vm1084	1.129.893	372.17%
vm1748	1.718.602	410.64%

Nombre del resolvedor	**Distancia Obtenida**	**% Dev. Valor óptimo**
Divide-and-conquer	357.995,84	306%
Divide-and-conquer + 2-opt	357.995,84	306%

Lecciones aprendidas

- Es mejor trabajar con estructuras más sencillas y convertirlas a la estructura deseada después de haber realizado todas las operaciones pertinentes.
- Es mucho más óptimo ejecutar instrucciones en línea que de forma paralela, ya que esto aumentaría la complejidad de nuestro código, y por consiguiente afectará su rendimiento.
- Este tipo de problemas trae muchas utilidades consigo pero a la vez es muy complejo y requiere un estudio a fondo